

## 版权注意事项：

- 1、书籍版权归作者和出版社所有
- 2、本PDF仅限用于个人获取知识，进行私底下的知识交流
- 3、PDF获得者不得在互联网上以任何目的进行传播
- 4、如觉得书籍内容很赞，请购买正版实体书，支持作者
- 5、请于下载PDF后24小时内删除本PDF。



拥有超过20年研发和管理经验的资深技术专家撰写  
系统、深入地讲解了在Nginx中使用Lua开发应用系统的技术和方法



李明江 著

*Nginx Lua in Action*

# Nginx Lua 开发实战



机械工业出版社  
China Machine Press

## 内 容 简 介

这是一部讲解如何在Nginx中使用Lua开发应用系统的实战类著作，作者是一位拥有超过20年研发经验的资深技术专家，内容的权威性毋庸置疑。

Nginx作为互联网应用系统中的核心服务，被广泛应用。Nginx通过配置可以实现负载均衡、反向代理等功能，还可以通过扩展开发更为复杂的业务逻辑。这其中，使用Lua语言开发是最方便和最流行的方法。本书以应用系统开发为主线，讲解了相关服务、模块和开发手册，并提供了大量真实的案例。

全书分为5个部分：

### 第一部分 Nginx基础篇（第1~5章）

首先，全面讲解了Nginx的基本操作，并讲解了MySQL、PostgreSQL、Redis、Memcached、MongoDB、OpenResty的基本操作；其次，分析了Nginx工作流程、核心技术以及架构。

### 第二部分 Lua脚本语言篇（第6~7章）

深入讲解了Lua脚本语言语法和Lua通用库，旨在帮助读者掌握Lua脚本语言，方便业务逻辑开发。

### 第三部分 Nginx开发技术篇（第8~10章）

讲解了Nginx应用系统开发中常用的相关知识，包括JSON格式、nginx.conf配置和Nginx下Lua实现机制。方便读者掌握开发过程中Nginx的配置和使用，同时了解Lua的实现机制，从而掌握在开发中使用Lua代码的方法。

### 第四部分 Nginx Lua开发实战篇（第11~26章）

讲解了Nginx下Lua常用模块以及示例代码，并提供了一个TCP私有服务器实例代码和一个WebSocket接入服务器实例代码。实战开发中，根据业务不同，会使用到非常多的模块，这部分内容讲解了常用的20多个模块，可以最大程度让读者节约查找资料的时间，还提供了2个示例程序，用于理解整个开发流程和技术使用方法。

### 第五部分 开发手册篇（第27~28章）

提供了ngx\_lua\_module模块配置命令详解和ngx\_lua函数详解。模块命令和API函数是开发中经常使用到的资料，用于查找函数说明和选择参数。

實戰



*Nginx Lua in Action*

# Nginx Lua 开发实战

李明江 著



机械工业出版社  
China Machine Press

## 图书在版编目 (CIP) 数据

Nginx Lua 开发实战 / 李明江著. —北京: 机械工业出版社, 2018.1  
(Web 开发技术丛书)

ISBN 978-7-111-59029-3

I.N… II. 李… III. 互联网络 - 网络服务器 - 程序设计 IV. TP368.5

中国版本图书馆 CIP 数据核字 (2018) 第 017367 号

## Nginx Lua 开发实战

出版发行: 机械工业出版社 (北京市西城区百万庄大街 22 号 邮政编码: 100037)

责任编辑: 何欣阳

责任校对: 殷虹

印 刷: 北京诚信伟业印刷有限公司

版 次: 2018 年 3 月第 1 版第 1 次印刷

开 本: 186mm×240mm 1/16

印 张: 36.25

书 号: ISBN 978-7-111-59029-3

定 价: 99.00 元

凡购本书, 如有缺页、倒页、脱页, 由本社发行部调换

客服热线: (010) 88379426 88361066

投稿热线: (010) 88379604

购书热线: (010) 68326294 88379649 68995259

读者信箱: hzit@hzbook.com

版权所有·侵权必究

封底无防伪标均为盗版

本书法律顾问: 北京大成律师事务所 韩光 / 邹晓东

## 为什么写这本书

在接手安防云任务之前，我做了近 10 年传统安防分布式平台的工作。安防云任务是基于云计算平台和 P2P 技术向用户提供 SaaS 服务。最初我的方案是将我们熟悉的分布式平台改造成云服务，进行互联网部署。

针对消费级摄像机的应用，我们对服务进行了裁剪，只留下几个服务器。但是需要几个资深的 C++ 服务器开发工程师提供技术支持。后来，新来的架构师经过一段时间的消化后，提出了 Nginx+Lua+Redis 架构。他一个人只花费一个多月的时间就把业务服务写好了。这件事情让我感受到了 Nginx+Lua 的魅力。

Nginx+Lua 架构带来的改变还远不止节约时间和成本。从做大型系统的角度来看，它还会带来更多的东西：

- **调试方便**：因为它不需要编译代码，相关访问模块是成熟稳定的，只需要调试新加的业务代码即可。大型系统特别是分布式系统，调试一个功能或代码的链条太长了，非常容易出错。
- **降低耦合**：因为架构的限制，代码只能在必需的阶段管理器中开发，代码是一个个 .lua 文件，耦合性大大降低。
- **框架良好**：因为先进的异步式多进程架构，可以充分利用系统资源。如果自行开发并维护这样一个框架，需要大量的人力、物力。
- **上手容易**：Lua 代码良好的结构和可读性，使其上手速度更快。团队成员经过快速培训就可以上手。

在学习和使用 ngx\_lua 的过程中我发现，网络上资料其实非常多，但是非常零散，没有整体性。虽然技术本身是很清晰和易用的，但是对于刚接触这门技术的开发者来说，想要有条

理、系统地把这些知识学完，可能会走一些弯路。因为很多资料已经过时了，甄别和调试会耗费很多时间和精力，同时查阅英文文档也比较花时间。于是，我就有了把自己的学习过程和心得整理成书的想法。

随后的时间里，我将自己学习 Nginx 下 Lua 开发的思路，以及这个过程资料一点点总结出来加以整理，终成本书。希望本书可以帮助跟我有一样需求的研发工程师快速了解并掌握 Nginx 下 Lua 开发技术。

## 本书的主要内容和特色

通常我们学习一门语言、一门技术的时候，都是这样一个过程：初学这项技术时，我们通常要知道 Lua 语言的细节，需要知道 Nginx 的结构；当我们着手学习时，首先需要搭建一个学习环境，以便了解系统的结构和运行机制，同时用于编写测试代码；运行第一个测试代码的时候，需要对 Nginx 进行配置，但是并不熟悉 Nginx 的配置，对于 `nginx.conf` 里的内容比较头疼；等到我们掌握了这项开发技术，开始业务系统开发了，又往往需要针对具体问题查找配置指令的用法和参数，以及查阅 API 的详细用法。

所以，本书按照这样的不同需求，对 Nginx 下 Lua 开发技术的不同关注点做了描述。

- ❑ Nginx 的基本知识，包含 Nginx 的使用、配置、安装、技术架构、技术特点、主要工作流程等。
- ❑ 外围关系型数据库、NoSQL 数据库、缓存等的使用范围、安装、使用方法、配置，如 MySQL、PostgreSQL、MongoDB、Redis、Memcached。
- ❑ Lua 语法详解，包含 Lua 系统库。
- ❑ Lua 常用库，包含 Redis、MySQL、Memcached、PostgreSQL、MongoDB、Bit、lfs、restry.http、lcurl、FFI、cjson、Template、WebSocket。
- ❑ 两个相对完整的 Lua 实例，结合实例以巩固涉及的知识点。
- ❑ ngx\_lua 的配置指令和 API，详细介绍了每一个官方指令和 API。

我希望无论是对于 Nginx Lua 的初学者，还是对于经验丰富的开发者，都可以通过本书学到基础知识，找到常用库的 API 说明，而不用查阅其他资料及官方英文文档。

## 本书面向的读者

本书适合初学 Nginx 下 Lua 开发的工程师使用。通过本书可以比较系统地学习 Lua 语言，



学习框架下经常用到的各组件使用方法，学习 Nginx 下 Lua 程序开发；可以参照本书的内容搭建学习环境，逐一测试各组件访问代码，并可通过后面的例子编写自己的访问代码和访问库。

本书适合有经验的 Nginx 和 Lua 开发工程师使用。通过本书可以快速查阅相关数据库、缓存、库的使用方法；可以查阅 Nginx 配置指令；还可以查阅 ngx\_lua 配置指令和 API。

本书同样适合做服务器端开发的资深工程师使用。书中讲解了 Nginx 的核心架构和主要的工作流程，也讲解了 Nginx 为了提高性能和并发所使用的一些关键技术，这些技术和算法对我们开发自己的高性能服务器有重要指导意义。

## 如何阅读本书

本书主要分五部分：

第一部分（第 1～5 章）介绍 Nginx 的基本操作，同时讲解了 MySQL、PostgreSQL、Redis、Memcached、MongoDB、OpenResty 的基本操作。通过对本部分的学习可以掌握这些服务的安装和使用方法，一般用于研发环境的搭建。这里还讲解了 Nginx 核心技术和工作流程，用于帮助读者进一步掌握 Nginx 的架构和流程。各个层次的读者都可以从本部分读起。

第二部分（第 6～7 章）详细讲解了 Lua 脚本语言和 Lua 通用库。学习 Lua 语言或查阅 Lua 语法的初学者可以直接阅读该部分相应章节。

第三部分（第 8～10 章）讲解了在 Nginx 开发中经常使用的一些技术，如 JSON 数据交换格式、nginx.conf 配置方法和配置指令；还讲解了 Nginx 下 Lua 开发的实现机制。学习 Nginx 配置的读者，学习和查阅 JSON 的读者可以直接阅读该部分相应章节，也可以跳过其他章节，直接学习 Nginx 下 Lua 的实现机制。

第四部分（第 11～26 章）详细介绍了常用 Lua 库和数据库等组件的使用方法，包括 Redis、MySQL、Memcached、PostgreSQL、MongoDB、Bit、lfs、resty.http、lcurl、FFI、cjson、Template、WebSocket。要了解和學習这些内容的读者，可以直接阅读对应章节。这里同时给出了两个 Lua 编程实例代码，要总体了解这项编程技术的读者可直接翻阅相应章节。

第五部分（第 27～28 章）详细介绍了 ngx\_lua 的配置指令和 ngx\_lua API，目的是帮助读者在工作中快速检索配置指令和 API。

## 勘误和资源

由于时间有限，技术能力有限，虽然已经尽量客观，在写作过程中排除自己主观的内容，

但难免有错误和不准确的地方，热忱希望你的批评和指正。

欢迎通过邮箱和我联系：13067722617@163.com。

## 致谢

首先需要感谢我的太太一直以来对我的信任和支持，她是我一路走来的动力。写这本书的时候，九哥已经 10 岁了，他已经知道我在做什么了，他以为我为荣，我也以他为傲。感谢我的丈人、丈母娘对我们的爱和一直以来的帮助。

写这本书的时候，我的父母又来到了我们的身边，为了和我们年轻人相处得更融洽，他们做出了巨大的改变。

最后要感谢出版社的杨福川和李艺，没有你们的努力是不会见到这本书的。感谢你们的支持和信任！



## Contents 目 录

### 前言

## 第一部分 Nginx 操作基础

### 第1章 Nginx高效服务器 ..... 2

- 1.1 Nginx 的特点 ..... 2
- 1.2 Nginx 的安装 ..... 3
- 1.3 configure 命令参数 ..... 7
- 1.4 小结 ..... 12

### 第2章 数据库的基本操作 ..... 13

- 2.1 Nginx 应用中的数据库 ..... 13
- 2.2 MySQL 关系型数据库 ..... 15
  - 2.2.1 yum 安装方法 ..... 16
  - 2.2.2 使用 mysql 测试服务 ..... 18
  - 2.2.3 MySQL 文件分布 ..... 19
  - 2.2.4 数据库操作 ..... 19
- 2.3 Redis 内存数据库 ..... 22
  - 2.3.1 Redis 安装 ..... 22
  - 2.3.2 启动 Redis 服务 ..... 23
  - 2.3.3 Redis 配置 ..... 23

- 2.3.4 参数说明 ..... 26

- 2.3.5 数据类型 ..... 29

### 2.4 PostgreSQL 关系型数据库 ..... 31

### 2.5 Memcached 内存数据库 ..... 33

- 2.5.1 Memcached 安装 ..... 34
- 2.5.2 连接编辑 ..... 34
- 2.5.3 管理 Memcached 服务 ..... 35
- 2.5.4 Memcached 命令 ..... 37

### 2.6 MongoDB 分布式 NoSQL 数据库 ..... 42

- 2.6.1 MongoDB 安装 ..... 43
- 2.6.2 mongod.conf 配置说明 ..... 45

### 2.7 小结 ..... 48

### 第3章 OpenResty ..... 49

- 3.1 OpenResty: 概述 ..... 49
- 3.2 OpenResty 的组成 ..... 50
- 3.3 OpenResty 的安装 ..... 52
- 3.4 Nginx 多实例 ..... 54
- 3.5 小结 ..... 54

### 第4章 Nginx核心技术 ..... 55

- 4.1 Nginx 设计目标 ..... 55

4.2	Nginx 架构 .....	57	6.1.3	安装 Lua 环境 .....	87
4.2.1	事件驱动 .....	57	6.2	Lua 基本语法 .....	88
4.2.2	异步多阶段处理 .....	59	6.2.1	第一个 Lua 程序 .....	88
4.2.3	模块化设计 .....	61	6.2.2	注释 .....	89
4.2.4	管理进程、工作进程设计 .....	63	6.2.3	标识符 .....	90
4.2.5	内存池 .....	65	6.2.4	关键词 .....	90
4.2.6	连接池 .....	66	6.2.5	全局变量 .....	90
4.2.7	时间缓存 .....	66	6.3	Lua 的数据类型 .....	90
4.2.8	延迟关闭 .....	67	6.4	Lua 变量 .....	96
4.2.9	跨平台 .....	67	6.4.1	赋值语句 .....	96
4.2.10	HTTP 模块管道过滤模式 .....	67	6.4.2	索引 .....	97
4.2.11	keepalive .....	68	6.5	Lua 循环 .....	98
4.2.12	pipeline .....	69	6.6	Lua 流程控制 .....	98
4.3	小结 .....	69	6.7	Lua 函数 .....	99
			6.7.1	函数的定义 .....	99
			6.7.2	多返回值 .....	100
			6.7.3	可变参数 .....	101
第5章	Nginx的工作流程 .....	70	6.8	Lua 运算符 .....	101
5.1	Nginx 的启动流程 .....	70	6.8.1	算术运算符 .....	102
5.2	管理进程的工作流程 .....	72	6.8.2	关系运算符 .....	102
5.3	工作进程的工作流程 .....	75	6.8.3	逻辑运算符 .....	103
5.4	配置加载流程 .....	76	6.8.4	其他运算符 .....	104
5.5	HTTP 框架初始化流程 .....	79	6.8.5	运算符的优先级 .....	105
5.6	HTTP 模块调用流程 .....	81	6.9	Lua 字符串 .....	106
5.7	HTTP 请求处理流程 .....	82	6.10	Lua 数组 .....	107
5.8	小结 .....	83	6.10.1	一维数组 .....	107
			6.10.2	多维数组 .....	108
			6.11	Lua 迭代器 .....	109
			6.11.1	泛型 for 迭代器 .....	109
			6.11.2	无状态的迭代器 .....	110
			6.11.3	多状态的迭代器 .....	111
			6.12	Lua 表 .....	112
<h2>第二部分 Lua 脚本语言</h2>					
第6章	Lua教程 .....	86			
6.1	Lua 基础 .....	86			
6.1.1	Lua 的特性 .....	86			
6.1.2	Lua 的应用场景 .....	87			

6.13	Lua 模块与包 .....	113
6.13.1	require 函数 .....	114
6.13.2	加载机制 .....	115
6.13.3	C 包 .....	115
6.14	Lua 元表 .....	116
6.14.1	_index 元方法 .....	117
6.14.2	_newindex 元方法 .....	118
6.14.3	为表添加运算符 .....	119
6.14.4	_call 元方法 .....	119
6.14.5	_tostring 元方法 .....	120
6.15	Lua 协同程序 .....	121
6.15.1	基本语法 .....	121
6.15.2	生产者-消费者问题 .....	124
6.16	Lua 错误处理 .....	125
6.16.1	语法错误 .....	125
6.16.2	运行错误 .....	125
6.16.3	错误处理 .....	126
6.16.4	error 函数 .....	126
6.16.5	pcall、xpcall、debug .....	127
6.17	Lua 调试 .....	128
6.18	Lua 垃圾回收 .....	130
6.19	Lua 面向对象 .....	131
6.19.1	Lua 中面向对象 .....	132
6.19.2	Lua 继承 .....	134
6.20	Lua 数据库访问 .....	136
6.21	小结 .....	137

## 第7章 Lua通用库 .....

7.1	字符串库 .....	138
7.2	表库 .....	141
7.3	文件 I/O 库 .....	143

7.3.1	简单模式 .....	144
7.3.2	完全模式 .....	145
7.3.3	其他方法 .....	146
7.4	数学库 .....	147
7.5	操作系统库 .....	150
7.6	小结 .....	151

## 第三部分 Nginx 开发技术

### 第8章 JSON数据交换格式 .....

8.1	什么是 JSON .....	154
8.2	JSON 转换为 JavaScript 对象 .....	155
8.3	JSON 与 XML 的比较 .....	155
8.4	JSON 语法规则 .....	156
8.5	格式化 .....	157
8.6	小结 .....	158

### 第9章 nginx.conf文件配置 .....

9.1	默认 nginx.conf 文件 .....	159
9.2	nginx.conf 示例 .....	162
9.3	全局配置与顶层配置块 .....	166
9.3.1	main 全局配置 .....	166
9.3.2	events 配置块 .....	170
9.3.3	http 服务器配置块 .....	172
9.3.4	ngx_http_core_module 变量 .....	194
9.3.5	stream .....	195
9.4	中文版 nginx.conf .....	201
9.5	小结 .....	204

### 第10章 Nginx下Lua实现机制 .....

10.1	ngx_lua 原理 .....	206
------	------------------	-----

10.2	HTTP 请求的处理阶段 .....	209	11.4.2	API 函数 .....	240
10.3	ngx_lua 的处理阶段 .....	210	11.4.3	技术点 .....	244
10.4	Lua 阶段解析 .....	212	11.4.4	问题列表 .....	246
10.4.1	init_by_lua .....	212	11.4.5	限制 .....	247
10.4.2	init_worker_by_lua .....	213	11.4.6	安装 .....	247
10.4.3	set_by_lua .....	214	11.5	小结 .....	247
10.4.4	rewrite_by_lua .....	216			
10.4.5	access_by_lua .....	217	<b>第12章</b>	<b>MySQL操作 .....</b>	<b>248</b>
10.4.6	content_by_lua .....	218	12.1	lua-resty-mysql 访问方式 .....	248
10.4.7	header_filter_by_lua .....	220	12.1.1	示例 .....	248
10.4.8	body_filter_by_lua .....	220	12.1.2	安装 .....	250
10.4.9	log_by_lua .....	220	12.1.3	方法与函数 .....	251
10.4.10	balancer_by_lua_block .....	221	12.1.4	多结果集返回示例 .....	254
10.5	小结 .....	222	12.1.5	其他注意事项 .....	255
			12.1.6	限制 .....	255
<b>第四部分 Nginx Lua 开发实战</b>			12.2	HttpDrizzleModule 访问方式 .....	255
<b>第11章</b>	<b>Redis操作 .....</b>	<b>224</b>	12.2.1	示例 .....	256
11.1	Redis 操作方法概述 .....	224	12.2.2	安装 .....	257
11.2	HttpRedis 访问方法 .....	225	12.2.3	技术点 .....	258
11.2.1	示例 .....	225	12.2.4	配置指令 .....	259
11.2.2	HttpRedis API .....	226	12.2.5	变量 .....	263
11.2.3	HttpRedis 变量 .....	228	12.2.6	输出格式 .....	264
11.3	HttpRedis2Module 访问方法 .....	229	12.3	HttpDrizzleModule 完整示例 .....	265
11.3.1	示例 .....	229	12.4	小结 .....	272
11.3.2	nginx.conf 配置 .....	230			
11.3.3	常用指令 .....	231	<b>第13章</b>	<b>Memcached操作 .....</b>	<b>273</b>
11.3.4	技术点 .....	234	13.1	mem-nginx-module 访问方式 .....	273
11.3.5	应答包解析 .....	238	13.1.1	概述 .....	273
11.4	lua-resty-redis 访问方法 .....	239	13.1.2	命令 .....	276
11.4.1	示例 .....	239	13.1.3	指令 .....	279
			13.1.4	安装 .....	281

13.1.5 说明 .....	281	16.4 说明 .....	307
13.1.6 示例 .....	282	16.5 小结 .....	308
13.2 lua-resty-memcached 访问 方式 .....	285	<b>第17章 lfs库的使用</b> .....	309
13.2.1 概述 .....	285	17.1 目录迭代示例 .....	309
13.2.2 API .....	286	17.2 安装 .....	310
13.2.3 自动日志 .....	291	17.3 LuaFileSystem 函数 .....	310
13.2.4 限制 .....	291	17.4 小结 .....	312
13.3 小结 .....	291	<b>第18章 resty.http库的使用</b> .....	313
<b>第14章 PostgreSQL操作</b> .....	292	18.1 安装 .....	313
14.1 概述 .....	292	18.2 概述 .....	314
14.2 配置指令 .....	293	18.3 函数 .....	315
14.3 配置变量 .....	295	18.3.1 连接类 .....	315
14.4 示例 .....	296	18.3.2 应答类 .....	318
14.5 小结 .....	298	18.3.3 代理类 .....	319
<b>第15章 MongoDB操作</b> .....	299	18.3.4 工具类 .....	319
15.1 安装 .....	299	18.4 小结 .....	320
15.2 配置 .....	299	<b>第19章 lcurl库的使用</b> .....	321
15.3 操作函数 .....	300	19.1 安装 .....	321
15.3.1 连接对象方法 .....	300	19.1.1 安装 libcurl .....	321
15.3.2 数据库对象方法 .....	301	19.1.2 安装 lcurl .....	322
15.3.3 列对象方法 .....	301	19.2 示例 .....	322
15.4 示例 .....	302	19.3 函数 .....	324
15.5 小结 .....	303	19.3.1 httpform 类 .....	325
<b>第16章 bit库的使用</b> .....	304	19.3.2 easy 类 .....	327
16.1 示例 .....	304	19.3.3 multi 类 .....	331
16.2 安装 .....	305	19.3.4 error 类 .....	333
16.3 函数 .....	305	19.3.5 share 类 .....	333
		19.4 常用变量 .....	334


19.4.1	字符串数组类选项 .....	334	21.2	函数 .....	354
19.4.2	字符串选项 .....	334	21.3	变量 .....	358
19.4.3	数值型选项 .....	336	21.4	小结 .....	358
19.4.4	布尔型选项 .....	337			
19.5	完整示例 .....	338	<b>第22章</b>	<b>lua-resty-template类的</b>	
19.6	小结 .....	340		<b>使用</b> .....	359
<b>第20章</b>	<b>FFI扩展C库</b> .....	341	22.1	示例 .....	359
20.1	示例 .....	341	22.2	模板符号 .....	360
20.1.1	调用外部 C 函数 .....	341	22.2.1	短转义符号 .....	361
20.1.2	使用 C 结构体数据 .....	342	22.2.2	上下文表中的复杂 key .....	361
20.2	FFI 库的使用 .....	344	22.2.3	HTML 转义 .....	361
20.2.1	载入 FFI 库 .....	344	22.2.4	保留的上下文 key 和 评论 .....	362
20.2.2	访问标准系统函数 .....	344	22.3	安装 .....	363
20.2.3	访问 zlib 压缩库 .....	345	22.3.1	Nginx/OpenResty 配置 .....	363
20.2.4	为一个 C 类型定义元 方法 .....	346	22.3.2	使用 document_root .....	363
20.2.5	转换 C 语法 .....	347	22.3.3	使用 template_root .....	364
20.3	FFI API .....	348	22.3.4	使用 template_location .....	364
20.3.1	声明和访问外部符号 .....	348	22.4	Lua API .....	364
20.3.2	创建 cdata 对象 .....	349	22.5	模板预编译 .....	368
20.3.3	C 类型信息 .....	349	22.6	模板助手 .....	368
20.3.4	功能函数 .....	350	22.7	用法示例 .....	369
20.3.5	特定目标信息 .....	351	22.7.1	引用模板 .....	369
20.3.6	方法回调 .....	351	22.7.2	Layouts 的 views .....	370
20.3.7	扩展标准库函数 .....	351	22.7.3	使用 Blocks .....	371
20.4	调用 curl 库的完整示例 .....	352	22.7.4	继承 .....	373
20.5	小结 .....	352	22.7.5	Macros .....	374
<b>第21章</b>	<b>cjson库的使用</b> .....	353	22.7.6	调用模板中的方法 .....	375
21.1	示例 .....	353	22.7.7	模板内嵌的 Angular 或其他 标签 / 模板 .....	376
			22.7.8	模板内嵌的 Markdown .....	376

22.7.9 LSP .....	377	<b>第25章 WebSocket接入服务器</b>	
22.8 FAQ .....	378	<b>实战</b> .....	413
22.9 小结 .....	379	25.1 nginx.conf 内容 .....	413
<b>第23章 WebSocket的使用</b> .....	380	25.2 ws_svr.lua 内容 .....	421
23.1 示例 .....	381	25.3 update_alerts 代码 .....	436
23.2 安装 .....	383	25.4 小结 .....	438
23.3 resty.websocket.server .....	383	<b>第26章 Nginx应用简述</b> .....	439
23.4 resty.websocket.client .....	386	26.1 简单系统 .....	439
23.5 resty.websocket.protocol .....	389	26.2 读写分离系统 .....	439
23.6 使用注意事项 .....	390	26.3 引入缓存系统 .....	440
23.7 小结 .....	390	26.4 缓存主从系统 .....	441
<b>第24章 TCP私有服务器实例</b> .....	391	26.5 小结 .....	442
24.1 协议 .....	391		
24.1.1 协议总体要求 .....	391	<b>第五部分 开发手册</b>	
24.1.2 包头定义 .....	392	<b>第27章 ngx_lua_module模块配置</b>	
24.1.3 协议命令 .....	393	<b>指令详解</b> .....	444
24.2 DDP 系统架构 .....	394	27.1 概述 .....	444
24.3 DDP 服务实现 .....	395	27.2 Lua 配置顺序 .....	456
24.3.1 nginx.conf 配置 .....	395	27.3 配置指令 .....	457
24.3.2 init.lua .....	398	27.4 小结 .....	487
24.3.3 ddp.lua .....	399	<b>第28章 ngx_lua API详解</b> .....	488
24.3.4 DDP 代码解析 .....	405	28.1 概述 .....	488
24.3.5 Redis 和 MySQL 的		28.2 API 与常量 .....	491
location .....	407	28.3 小结 .....	565
24.3.6 管理页面 REST 操作 .....	411		
24.4 小结 .....	412		



## 第一部分 *Part 1*

# Nginx 操作基础

- 第 1 章 Nginx 高效服务器
  - 第 2 章 数据库的基本操作
  - 第 3 章 OpenResty
  - 第 4 章 Nginx 核心技术
  - 第 5 章 Nginx 的工作流程
- 



# Nginx 高效服务器

Nginx 是一款轻量级的 Web 服务器，特点是高性能、高并发。它由俄罗斯程序设计师 Igor Sysoev 开发，供俄国大型入口网站及搜索引擎 Rambler 使用。Nginx 在 BSD-like 协议下发行，是一款高性能 Web 服务器，目前在 Web 服务器中排名第二。虽然 Apache 还是全球 Web 服务器的“老大”，但是 Nginx 已经占到了 Web 服务器市场 22% 以上的份额，是成长最快的 Web 服务器。Nginx 使用了大量的高并发和低内存占用技术，并使用了高可靠性技术，拥有高过 Apache 一个数量级以上的接入能力。因为并发能力强的特点，Nginx 在中国的互联网公司中得到了大量应用，中国的大型互联网公司无一不使用了 Nginx，以应对中国众多的网民，以及各种抢购热潮（如“双十一”）、世界杯等热点事件。Nginx 在这种大量的流量涌入、需要分流、导流、反向代理的场合下得到大量应用。

## 1.1 Nginx 的特点

与其他 Web 服务器相比，Nginx 具有以下显著特点。

### 1. 速度更快

Nginx 使用了预读、连接池、内存池等技术，使得单次 HTTP 请求速度更快。同时，因为其整体的多进程架构以及轻任务思想，在更多连接的情况下（以万为单位的并发情况下），Nginx 比其他 Web 服务器速度更快。

### 2. 扩展性好

Nginx 的结构是“核心 + 模块”的结构，Nginx 本身就是一个基于 Epoll 或 Kqueue 的事件处理和分发架构，管理 HTTP 主流程，其他功能都可以通过模块实现。模块专注于自

身功能实现，可以更稳定，模块的升级和修复不影响其他功能以及核心本身。模块可以不断添加或升级，如事件（event）模块、代理模块、过滤模块、请求地址获取模块、地址转换模块、应答处理模块、日志模块等。Nginx 提供了众多的模块以供选择，可以配置出不同行为的 Web 服务器。

Nginx 提供了 C 级别的模块开发机制，但 C 级别的开发需要遵从复杂的数据结构。现在可以通过 ngx\_lua 模块以 Lua 脚本实现业务逻辑。得益于 Lua 协程的支持，ngx\_lua 在万级并发请求时只占用很少的内存，而性能都是万级（Operation Per Second，每秒操作次数），这使得 Nginx 的扩展性更好。

### 3. 高可靠性

得益于整体架构的优秀以及模块设计的简单性，Nginx 拥有极高的可靠性，在各大型网站中得到了认可。Nginx 核心由一个任务很轻的管理进程（master 进程）和若干工作进程（worker 进程）组成。具体的 HTTP 请求在工作进程内负载均衡，如果某个工作进程异常终止了，管理进程会迅速重启一个新的工作进程接替该进程。

### 4. 低内存占用

一般情况下，10000 个非活跃的 HTTP 保活连接仅占用 2.5MB 内存。而 ngx\_lua 每扩展 10000 个连接也仅占 2.xMB 内存，使得 Nginx 可以大量部署。

### 5. 高并发能力

一般 Nginx 是部署在万级以上的场合下。为了应付海量的请求，各网站都需要单机能处理峰值 10 万以上并发请求的 Web 服务器。理论上，Nginx 处理能力的上限仅受内存限制，简单的业务场景下 Nginx 还可以提供更高的处理能力。

Nginx 全异步、非阻塞 I/O 的思想贯穿在核心、模块以及 ngx\_lua 模块中，无论是自己实现的模块，还是通过 Lua 实现的脚本代码，都是非阻塞地高速运行。

### 6. 热部署

因为 Nginx 的管理进程和工作进程是分开设计的，所以可以实现热部署功能，即能在系统不间断的情况下升级可执行程序、更新配置文件、更新日志文件等。

### 7. 开源

Nginx 遵守相对自由的 BSD 协议。用户可以自由使用 Nginx，还可以自由修改和使用 Nginx 的源码。用户可以在节省大量时间和成本的情况下，得到一个高性能的服务器框架。

## 1.2 Nginx 的安装

Nginx 有预编译版本、源码，因为我们进行的是 Nginx 配套的开发工作，考虑到功能、性能等原因，源码安装方法更适合本书读者，故这里主要介绍源码安装方法。

另外，有一个 OpenResty 项目将 Nginx 整合进 Lua、LuaJIT 和其他配合组件，形成一个多功能的开发型 Web 服务器套件，因为我们学习 Lua 开发，所以推荐使用 OpenResty，本章介绍的 Nginx 安装方法可供读者自定义系统时参考。

因为典型的互联网应用基本安装在 Linux 上，所以，本文中的安装和编译均以 Linux 为例，系统为 CentOS，用其他 Linux 发布包将对应工具换下即可。

### 1. 下载 Nginx 源文件

安装文件与源码可以从 Nginx 官网下载，地址为 <http://nginx.org/en/download.html>。

我们需要下载源码，本书以 1.10.0 版本为例。

下载：

```
wget http://nginx.org/download/nginx-1.10.0.tar.gz
```

解压：

```
tar -zxvf nginx-1.10.0.tar.gz
```

### 2. 检查安装依赖项

要使用 Nginx 常用功能，首先需要确保操作系统上至少安装如下软件。

1) GCC (GNU Compiler Collection)：用来编译 C 语言程序。在使用源码方式安装 Nginx 时，需要使用 GCC 编译 Nginx 及后面要用到的模块源码。

2) PCRE (Perl Compatible Regular Expressions, Perl 兼容正则表达式)：由 Philip Hazel 开发的函数库，目前为很多软件使用，支持正则表达式，由 RegEx 发展而来。ngx\_lua 中的 ngx\_re.\* 系列 API 需要使用 PCRE 库。如果在 nginx.conf 中使用了正则表达式，那么编译 Nginx 时就必须把 PCRE 库编译进 Nginx，Nginx 中的 HTTP 模块要靠它解析正则表达式。通常情况下都会用到正则表达式。

3) Zlib 库：用于对 HTTP 报文内容做 gzip 格式压缩，如果在 nginx.conf 中配置了 gzip on，并指定某些 Content-Type 的 HTTP 应答包体使用 gzip 进行压缩，以减小网络传输量，那么就需要在编译 Nginx 时指定 zlib，将其编译进 Nginx。

4) OpenSSL 库：使用 HTTPS 在 SSL 上传输 HTTP，就需要安装 OpenSSL。另外，在 ngx\_lua 中使用 MD5、SHA1 等散列函数时，也需要安装 OpenSSL。

Nginx 的特点是高性能和定制灵活，实现多功能的 Web 服务，所以各种模块可以根据需求组合。这些模块可能使用到其他支撑的基础库，因此需要保证这些库的正确安装，上面列出的 4 个库就是基础库，如果用到了其他库，也需要首先安装，具体需求参见模块的相关文档和手册。

在 Nginx 上安装工具的方法很多，我们可以使用 yum 安装，相对比较简单，命令如下：

```
yum -y install gcc pcre pcre-devel zlib zlib-devel openssl openssl-devel
```

也可以分步骤安装，命令如下：

```

yum install -y gcc
yum install -y gcc-c++
yum install -y pcrc pcrc-devel
yum install -y openssl openssl-devel

```

### 3. Linux 内核参数优化

默认的 Linux 内核参数适合于通用的场景。Linux 的特点是可以根据不同的应用场景调整内核参数以及安装的包和工具，使之成为专用的服务器。通常 Linux 用作各种服务器更为合适，作为高性能的 Web 服务器的基础服务是比较典型的应用。

当 Nginx 作为静态 Web 服务器、反向代理服务器等不同服务器时，所需的内核参数是不同的，本文针对通常的多并发 Nginx 应用，给出内核参数修改样表。

修改 /etc/sysctl.conf，常用配置如下：

```

fs.file-max = 999999
net.core.netdev_max_backlog = 8096
net.core.rmem_default = 262144
net.core.wmem_default = 262144
net.core.rmem_max = 2097152
net.core.wmem_max = 2097152
net.ipv4.tcp_tw_reuse = 1
net.ipv4.tcp_keepalive_time = 600
net.ipv4.tcp_fin_timeout = 30
net.ipv4.tcp_max_tw_buckets = 5000
net.ipv4.ip_local_port_range = 1024 61000
net.ipv4.tcp_rmem = 4096 32768 262142
net.ipv4.tcp_wmem = 4096 32768 262142
net.ipv4.tcp_syncookies = 1
net.ipv4.tcp_max_syn_backlog = 1024

```

执行 sysctl -p 命令，使修改生效。

各内核参数的作用如下。

1) fs.file-max：表示进程（在 Nginx 里指一个工作进程）可以同时打开的最大句柄数。本参数影响最大并发连接数。

2) net.ipv4.tcp\_syncookies：解决 TCP 的 SYN 攻击。

3) net.ipv4.tcp\_tw\_reuse：参数为 1 时表示允许将 TIME\_WAIT 状态的套接字重新用于新的 TCP 连接。服务器上的 TCP 协议栈在工作时会有大量的 TIME\_WAIT 状态连接，重新使用这些连接对于服务器处理大并发连接非常有用。

4) net.ipv4.tcp\_keepalive\_time：表示当 keepalive 启用时，TCP 发送 keepalive 消息的频率。默认为 2 小时，如果本值变小，可以更快地清理无效的连接。

5) net.ipv4.tcp\_fin\_timeout：表示服务器主动关闭连接时，套接字的 FIN\_WAIT-2 状态最大时间。

6) net.ipv4.tcp\_max\_tw\_buckets：表示操作系统允许的 TIME\_WAIT 套接字数量的最大值。当超过这个值，TIME\_WAIT 状态的套接字被立即清除并输出警告消息，默认值为 180000，过多的 TIME\_WAIT 套接字会使服务器速度变慢。

7) `net.ipv4.tcp_max_syn_backlog` : 表示 TCP 三次握手阶段 SYN 请求队列最大值, 默认为 1024。调置为更大的值可以使 Nginx 在非常繁忙的情况下, 若来不及接收新的连接时, Linux 不至于丢失客户端新创建的连接请求。

8) `net.ipv4.ip_local_port_range` : 定义 UDP 和 TCP 连接中本地端口范围 (不包括连接到远端的端口)。

9) `net.ipv4.tcp_rmem` : 定义 TCP 接收缓存 (TCP 接收窗口) 的最小值、默认值、最大值。

10) `net.ipv4.tcp_wmem` : 定义 TCP 发送缓存 (TCP 发送窗口) 的最小值、默认值、最大值。

11) `net.core.netdev_max_backlog` : 当网卡接收报文速度大于内核处理速度时, 本参数设置这个缓冲队列最大值。

12) `net.core.rmem_default` : 表示内核套接字接收缓冲区默认值。

13) `net.core.wmem_default` : 表示内核套接字发送缓冲区默认值。

14) `net.core.rmem_max` : 表示内核套接字接收缓冲区最大值。

15) `net.core.wmem_max` : 表示内核套接字发送缓冲区最大值。

#### 4. 配置安装选项

通常 Nginx 安装在 `/opt` 或 `/usr/local` 目录下。其他配置选项根据需要进行选择, 具体意义和使用方法参见 1.3 节中各个配置项分类表格, 或使用 `./configuration -help` 命令查看。对于具体的 `ngx_lua` API 使用中所需要的参数, 也可以在 API 章节查看 API 描述。

配置命令如下:

```
./configure --prefix=/opt/nginx --sbin-path=/opt/nginx/sbin/nginx
```

#### 5. 编译与安装

命令如下:

```
make
make install
```

未在 `./configuration` 中指定目录的情况下, Nginx 默认会安装到 `/usr/local/nginx` 目录下。安装后的 Nginx 可以通过复制创建新实例, 以方便调试和开发。

#### 6. 启动、停止、重启

启动命令如下:

```
/opt/nginx/sbin/nginx -p /opt/nginx/
```

如果在编译的时候通过 `--prefix` 和 `--sbin-path` 指定了目录, 可以直接使用 Nginx 启动。

`-p` 指定 Nginx 的目录。因为同一台服务器可以运行多个 Nginx 实例, 所以需要指定当前实例的目录。这个参数将影响很多 `ngx_lua` API 中文件参数的检索。通常将配置文件等均放在指定目录下。

多个 Nginx 实例可以编译不同的参数和模块，以实现不同的应用。

在本机浏览器中输入 `http://127.0.0.1` 或在其他机器上输入 Nginx 所在服务器 IP，如果看到了“Welcome to nginx”页面，表示 nginx 启动成功。

停止命令如下：

```
/opt/nginx/sbin/nginx -p /opt/nginx -s stop
```

如果编译时指定了 Nginx 的工作目录，可以直接使用 `nginx -s stop` 启动。

重启（重新载入配置文件）命令如下：

```
/opt/nginx/sbin/nginx -p /opt/nginx -s reload
```

如果编译时指定了目录，可以直接使用 `nginx -s reload` 重启。

### 1.3 configure 命令参数

使用 configure 命令参数可以在新编译的 Nginx 程序里打包指定的模块，或去除指定的模块，这样可以自定义 Nginx 功能，同时可以减少内存占用。下面介绍常用的 configure 命令参数。

#### 1. 编译参数

编译器相关参数如表 1-1 所示。

表 1-1 编译器相关参数

参数	说明
<code>--with-cc=PATH</code>	C 编译器路径
<code>--with-cpp=PATH</code>	C++ 编译器路径
<code>--with-cc-opt=OPTIONS</code>	链接时使用到第三方库时，需要指定链接参数。如需要使用 hello 库，则可做以下设置： <code>--with-cc-opt=hello -L/usr/local/hello</code>
<code>--with-cpu-opt=CPU</code>	指定 CPU 处理器架构，取值范围为 <code>pentium</code> 、 <code>pentiumpro</code> 、 <code>pentium3</code> 、 <code>pentium4</code> 、 <code>athlon</code> 、 <code>opteron</code> 、 <code>sparc32</code> 、 <code>sparc</code> 、 <code>ppc64</code>

#### 2. 路径参数

configure 支持的路径参数如表 1-2 所示。

表 1-2 configure 支持的路径参数

参数	说明	默认值
<code>--prefix=PATH</code>	Nginx 安装部署的根目录	默认为 <code>/usr/local/nginx</code> 目录。可以通过命令行添加 <code>-p</code> 参数动态修改
<code>--sbin-path=PATH</code>	可执行文件放置路径	<code>&lt;prefix&gt;/sbin/nginx</code> 如果设置 <code>--prefix=/usr/local/nginx</code> ，则 Nginx 运行文件在 <code>/usr/local/nginx/sbin/nginx</code>
<code>--conf-path=PATH</code>	配置文件存放路径	<code>&lt;prefix&gt;/conf/nginx.conf</code>



(续)

参数	说明	默认值
--error-log-path=PATH	error 错误日志路径，是 error.log 文件路径。可以在 nginx.conf 中配置成不同请求的日志，保存到不同的 log 文件中	<prefix>/log/error.log
--pid-path=PATH	pid 文件存放路径。pid 文件存放以 ASCII 码存放的 Nginx 管理进程 ID。这个 ID 在通过命令行向进程发送命令时使用，而不是通过使用 /usr/local/nginx/sbin/nginx -p /usr/local/nginx -s stop 停止 Nginx	<prefix>/logs/nginx.pid 可以在 nginx.conf 中修改 pid 的目录和文件名。 假设 pid 文件路径为 /data/logs/nginx.pid，则命令如下： kill -QUIT 'cat /data/logs/nginx.pid'
--lock-path=PATH	lock 文件存放路径	<prefix>/logs/nginx.lock
--buildidir=DIR	configure 执行与编译期间产生的临时文件存放目录	<nginx source path>/objs
--with-per_modules_path	perl 模块路径。指定第三方 perl 模块	无
--with-perl=PATH	perl 二进制文件路径。如果配置的 Nginx 要执行 perl 脚本，则需要配置此目录	无
--http-log-path=PATH	access 日志目录，每一个 HTTP 请求处理结束都会记录此日志	<prefix>/log/access.log
--http-proxy-temp-path=PATH	Nginx 作为 HTTP 反向代理时，上游服务器产生的 HTTP 包体需要临时存放在磁盘文件时，将保存在本配置项下的目录中	<prefix>/proxy_temp 一般为了性能，通常将此目录设置为内存虚拟磁盘
--http-fastcgi-temp-path=PATH	FastCGI 使用的临时文件目录	<prefix>/fastcgi_temp
--http-uwsgi-temp-path=PATH	uWSGI 使用的临时文件目录	<prefix>/uwsgi_temp
--http-scgi-temp-path=PATH	SCGI 使用的临时文件目录	<prefix>/scgi_temp

3. 依赖参数

依赖参数如表 1-3 所示。

表 1-3 依赖参数

参数	说明
--without-pcre	如果确认不使用正则表达式，那么使用本参数取消正则表达式支持
--with-pcre	强制使用 PCRE 库
--with-pcre = DIR	指定 PCRE 库源码路径，编译 Nginx 时会进入该目录编译 PCRE 源码
--with-pcre-opt=OPTIONS	编译 PCRE 源码时需要加入的编译选项
--with=libatomic	强制使用 atomic 库。atomic 库是 CPU 架构独立的一种原子操作，支持下列体系：x86（包括 i386 和 x86_64）、PPC64、Sparc64（V9 或更高版本）或者安装了 GCC4.1.0 及以上版本
--with-libatomic=DIR	atomic 库所在的位置
--with-zlib=DIR	指定 zlib 库源码目录，编译 Nginx 时会自动进入该目录编译源码

(续)

参数	说明
--with-zlib-opt=OPTIONS	编译 zlib 源码时希望加入的编译选项
--with-zlib-asm=CPU	指定汇编优化功能, 目前支持两种架构: pentium 和 pentiumpro
--with-MD5=DIR	指定 MD5 库源码位置, 编译 Nginx 时会进入该目录编译 MD5 源码。注意, Nginx 源码中已经实现了 MD5 算法, 如果没有特殊要求, 不要指定外在的 MD5 库
--with-MD5-opt=OPTIONS	编译 MD5 时需要加入的选项
--with-MD5-asm	使用 MD5 的汇编源码
--with-SHA1=DIR	指定 SHA1 库源码位置, 编译时会进入目录自动编译。注意, OpenSSL 中包含了 SHA1 算法, 如果安装了 OpenSSL, 那么可以使用 OpenSSL 的算法
--with-SHA1-opt=OPTIONS	编译 SHA1 源码时希望加入的编译选项
--with-SHA1-asm	使用 SHA1 的汇编源码

#### 4. 模块参数

Nginx 的架构分为核心代码 (Nginx core) 和功能模块 (module), 以模块形式实现灵活而强大的功能。模块根据需求灵活使用, 需要在 configure 阶段把需要用到的模块加载到 Nginx 中来。

Nginx 的模块大致可以分为核心事件 (event) 模块、默认会编译进 Nginx 的 HTTP 模块、默认不会编译进 Nginx 的 HTTP 模块和其他模块。

Nginx 的核心事件模块相关参数如表 1-4 所示。

表 1-4 Nginx 的核心事件模块相关参数

参数	说明
--with-rtsig_module	使用 rtsig 模块处理事件驱动
--with-select_module	使用 select 模块处理事件驱动。select 是常用的多路复用机制。默认情况下不安装 select 模块, 除非找不到其他更好的模块
--without-select_module	不安装 select_module
--with-poll_module	使用 poll 模块处理事件驱动 poll 性能与 select 差不多, 远不如 epoll 模块。默认情况下不安装 poll 模块
--with-aio_module	使用 AIO 方式处理事件驱动 注意, AIO 只能与 FreeBSD 上的 kqueue 事件处理机制合作, Linux 上无法使用

默认会编译进 Nginx 的 HTTP 模块如表 1-5 所示。

表 1-5 默认会编译进 Nginx 的 HTTP 模块

参数	说明
--without-http_charset_module	不安装 http_charset_module。本模块可以将服务器发出的 HTTP 响应重新编码
--without-http_gzip_module	不安装 http_gzip_module。本模块可以按照配置文件指定的 Content-Type 对特定大小的 HTTP 响应包体进行 gzip 压缩
--without-http_ssi_module	不安装 http_ssi_module。本模块可以在向客户端返回的 HTTP 包体中加入特定的内容, 如加入 HTML 文件中固定的页头和页尾



(续)

参数	说明
--without-http_userid_module	不安装 http_userid_module。本模块可以通过 HTTP 请求头部中一些字段认证用户信息, 确定请求是否合法
--without-http_access_module	不安装 http_access_module。本模块可以根据 IP 地址限制客户端对服务器的访问
--without-http_auth_basic_module	不安装 http_auth_basic_module。本模块可以提供最简单的用户名 / 密码认证
--without-http_autoindex_module	不安装 http_autoindex_module。本模块提供简单的目录浏览功能
--without-http_geo_module	不安装 http_geo_module。本模块定义一些常量, 用于与客户端 IP 地址关联, Nginx 针对不同地区客户端返回不一样的结果。例如, 不同地区显示不同语言的网页
--without-http_map_module	不安装 http_map_module。本模块可以建立一个 key/value 映射表, 不同的 key 得到相应的 value, 这样可以针对不同的 URL 做特殊处理。例如, 返回 302 重定向响应时, 可以根据 URL 不同返回不同的 location
--without-http_split_client_module	不安装 http_split_client_module。本模块根据客户端信息如 IP、header 头、cookie 等进行区分处理
--without-http_referer_module	不安装 http_referer_module。本模块可以根据请求中的 referer 字段拒绝请求
--without-http_rewrite_module	不安装 http_rewrite_module。本模块提供 HTTP 请求的重定向功能, 依赖 PCRE 库
--without-http_proxy_module	不安装 http_proxy_module。本模块提供基本的 HTTP 反向代理功能
--without-http_fastcgi_module	不支持 http_fastcgi_module。本模块提供 FastCGI 功能
--without-http_memcached_module	不安装 http_memcached_module。本模块可以直接从 Memcached 服务器中读取数据并返回给客户端
--without-http_limit_zone_module	不安装 http_limit_zone_module。本模块限制某个 IP 地址并发连接数。例如, 对一个 IP 限制只允许一个连接
--without-http_limit_req_module	不安装 http_limit_req_module。本模块限制某个 IP 地址并发请求数
--without-http_empty_gif_module	不安装 http_empty_gif_module。本模块使 Nginx 收到无效请求时, 返回内存 1×1 像素的 GIF 图片, 可以节省系统资源
--without-http_browser_module	不安装 http_browser_module。本模块根据 HTTP 请求中的 user-agent 字段识别浏览器
--without-http_upstream_ip_hash_module	不安装 http_upstream_ip_hash_module。本模块用于在 Nginx 与后端服务器连接时, 根据 IP 做散列运算决定与哪台服务器通信, 实现可以保持会话的负载均衡

默认不会编译进 Nginx 的 HTTP 模块如表 1-6 所示。

表 1-6 默认不会编译进 Nginx 中的 HTTP 模块

模块	说明
--with-http_ssl_module	安装 http_ssl_module。让 Nginx 支持 SSL 协议, 实现 HTTPS 连接
--with-http_realip_module	安装 http_realip_module。本模块从客户端请求的头域信息 (如 X-Real-IP 或 X-Forwarded-For) 获取真正的客户端 IP

(续)

模块	说明
--with-http_addition_module	安装 http_addition_module。在返回给客户端的 HTTP 包头或包体尾部追加内容
--with-http_image_filter_module	安装 http_image_filter_module。将符合配置的图片实时压缩为指定大小的缩略图再发送给用户。支持 JPEG、PNG、GIF 格式。注意, 本模块依赖于 ligd 库, 需要在系统内首先安装 ligd
--with-http_xslt_module	安装 http_xslt_module。可以在将 XML 格式数据发送给客户端之前加入 XSL 渲染
--with-http_sub_module	安装 http_sub_module。可以将返回给客户端的 HTTP 应答包中指定的字符串替换为需要的字符串
--with-http_geoip_module	安装 http_geoip_module。可以依据 MaxMind GeoIP 的 IP 地址数据库分析得到客户端的实际物理地址。本模块依赖于 MaxMind GeoIP 库文件
--with-http_flv_module	安装 http_flv_module。可以在向客户端发送 HTTP 响应时, 对 FLV 格式视频文件头部做一些处理, 使客户端可以正常观看、拖动
--with-http_dav_module	安装 http_dav_module。让 Nginx 支持 Webdav 标准, 如支持 Webdav 中的 put、delete、copy、mkcol、move 等请求
--with-http_gzip_static_module	安装 http_gzip_static_module。可以在做 gzip 压缩前首先检查是否已经存在经过 gzip 压缩的 gz 文件, 如果有则直接返回。提前在服务器上压缩好文档, 以减少压缩带来的系统开销
--with-http_mp4_module	安装 http_mp4_module。使客户端可以观看、拖动 MP4 视频
--with-http_random_index_module	安装 http_random_index_module。在客户端访问某个目录时, 随机返回该目录下的任意文件
--with-http_secure_link_module	安装 http_secure_link_module。提供一种验证请求是否有效的机制。例如, 验证 URL 中需要加入的 token 参数是否属于特定客户端
--with-http_sub_status_module	安装 http_sub_status_module。提供 Nginx 性能统计页面
--with-http_degradation_module	安装 http_degradation_module。本模块针对一些特殊的系统调用做优化, 但暂时不支持 Linux 系统
--with-google_perftools_module	安装 google_perftools_module。提供 Google 性能测试工具

其他 Configure 参数如表 1-7 所示。

表 1-7 其他 configure 参数

参数	说明
--user=USER	声明 Nginx 工作进程运行时所属用户。不要将工作进程所属用户设成 root。工作进程用户级别低于管理进程用户以便于管理, 在工作进程意外中止时, 管理进程可以正常启动工作进程
--group=GROUP	设置工作进程运行时所属的组
--with-debug	将 debug 级别的日志编译进 Nginx。也可以在配置文件中修改日志级别
--without-http	禁用 HTTP 服务器
--without-http-cache	禁用 HTTP 服务器里的缓存 Cache 特性
--with-ipv6	支持 IPV6
--with-file-aio	启用文件异步 I/O 功能处理磁盘文件, 需要 Linux 支持异步 I/O

(续)

参数	说明
--add-module=PATH	在 Nginx 里加入第三方模块时, 指定第三方模块的路径
--with-mail	安装邮件服务器反向代理模块, 使 Nginx 可以反射代理 IMAP、POP3、SMTP 等协议
--with-mail_ssl_module	安装 mail_ssl_module。基于 SSL/TLS 协议使用邮件, 依赖于 OpenSSL 库
--without-mail_pop3_module	不安装 mail_pop3_module。使用 --with-mail 参数后, 本模块默认安装
--without-mail_imap_module	不安装 mail_imap_module。使用 --with-mail 参数后, 本模块默认安装
--without-mail_smtp_module	不安装 mail_smtp_module。使用 --with-mail 参数后, 本模块默认安装
--with-stream_core_module	安装 ngx_stream_core_module。从 1.9.0 开始加入, 支持 TCP 处理能力或负载均衡, 默认不加入, 使用本参数打开

## 1.4 小结

本章介绍了 Nginx 服务器的特点、以源码方式编译和安装 Nginx 的方法、Nginx 常用操作, 并分类别介绍了 configure 命令参数。

## 数据库的基本操作

Nginx 应用系统通过以 Nginx 为核心，合理搭配 Redis、Memcached、MySQL、PostgreSQL、MongoDB 等服务器，可起到数据内存缓存、内存数据库、关系型数据库、NoSQL 数据库等作用。互联网系统上有各种关系型数据需要存储，需要用到关系型数据库。互联网要应对大量的高并发请求，就需要高速处理 HTTP 请求，需要用到各种数据缓存、页面缓存、操作缓存等。同时互联网上有大量的结构化非关系型数据要存储，还有各类音 / 视频、图片要缓存，需要用到各类内存型数据库、NoSQL 数据库等，以形成完整的应用。所以本章将从这些数据库、缓存产品的作用、特点、安装方法、常用命令及配置文件解析展开。

### 2.1 Nginx 应用中的数据库

在互联网公司，Nginx 基本是标配组件，主要场景是负载均衡、反向代理、代理缓存、限流等场景，而把 Nginx 作为一个 Web 容器使用还不是那么广泛。但是因为 Nginx 的二次开发性非常好，所以很多公司会以 Nginx 为核心开发业务系统，在系统中容纳各种服务和组件，组成一个开放的、易于扩展的系统，以实现高容量和分布式的架构。

通常，互联网上的系统是分布式、集群式，由若干功能相同的服务器组成集群，以响应大并发请求。分布式系统提供并行处理能力和内存型数据缓存或存储功能。目前流行的架构是 Hadoop，Map-Reduce 式的并行处理架构提供了并行处理能力，一个任务被分解成多个 Map 处理任务，同时被多个系统或主机处理，结果经 Reduce 转换成用户响应。分布式存储也是类似的机制，互联网上的数据大量被存储在廉价主机的内存中，在根服务的调度下响应用户请求。这就需要用到各种关系型数据库存储关系型数据，用到内存缓存以缓

存图片、音频、视频等数据，用到内存数据库以缓存和存储各种 NoSQL 和 SQL 数据。不同的应用使用不同的数据组件，以形成需要的应用功能。

一个常见的 Nginx 应用如图 2-1 所示。一个典型的应用可以使用 Nginx 作为负载均衡器或其他负载均衡器，将请求按负载均衡算法调度到后端的 Nginx 应用服务器上。Nginx 使用 Redis 作为数据缓存，使用 Memcached 作为文件缓存，使用 MongoDB 持久化 NoSQL 数据，使用 MySQL 集群作为关系型数据库。Nginx 还可以给其他类似 Java 或 PHP 服务做反向代理服务器或 CGI 缓存数据库。

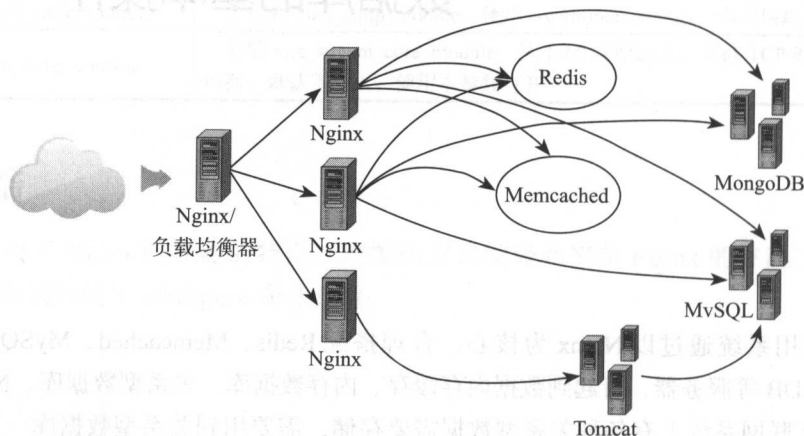


图 2-1 Nginx 应用

以 Nginx+Lua 为核心的架构，可以使用 Lua 语言作为“胶水”，“黏合”各种流行数据库组件，在 Lua 的表达能力下形成各种复杂的应用。本章分别介绍常用的各种数据组件的主要特点、安装、部署和常用操作，以方便学习和使用这些组件，运营级的安装和配置请自行深入研究。根据尽量保证研发和使用一致性的原则，本文只介绍 Linux 下各组件的安装和使用方法，因为主流的系统都是基于 Linux 开发的。

各数据组件都有其定位和应用场景，理解各组件、合理地应用组件可以大大降低开发工作的难度，并实现大型的应用和系统。

- MySQL 是关系型数据库，用于存储关系型数据。MySQL 支持读写分区，可以通过代理分离读和写操作，实现高性能。在读写分离的机制下，一个 MySQL 簇由代理服务器、主服务器和从服务器构成。主服务器负责写入，多实例的从服务器负责读响应，主、从服务器之间通过数据同步/异步地写入数据。各服务器可以动态扩展以增容，自然地实现了备份。通过合理的配置，可以实现相当量级的并发访问。MySQL 在大型互联网系统中得到广泛使用。基于读写分区的簇，在业务上再进行垂直分区，则可实现大型、超大型系统。在簇内还有分库、分表等技术，这些技术可以实现库内的大型数据存储。
- Redis 是一个流行的内存数据库，与 Memcached 相似，但支持将内存的数据持久化

到本地文件中，而且支持更多的数据种类，不仅仅支持简单的 key/value 类型的数据，同时支持 list、set、hash 等数据结构的存储。内存数据库解决了互联网上大并发和高速访问的问题。数据存放在内存中读写比从磁盘中读写速度快两个数量级。通常，在互联网系统中会将 Redis/Memcached 与 MySQL/PostgreSQL 组合使用，解决高速访问和数据持久化的问题。

- PostgreSQL 是对象关系型数据库，支持大部分 SQL 标准，并且提供了许多其他现代特性：复杂查询、外键、触发器、视图、事务完整性、MVCC。PostgreSQL 可以用许多方法扩展，例如，通过增加新的数据类型、函数、操作符、聚集函数、索引。PostgreSQL 是支持数据种类最多的数据库，支持的接口也最丰富。PostgreSQL 在数据库总的市场上占有的份额还不小，但上升很快。现在的云主机厂商越来越多地提供 PostgreSQL 数据库。
- Memcached 是一个内存型数据库，用在动态 Web 应用上以减轻数据库负载。通过在内存中缓存数据和对象来减少读取数据库的次数，从而提高访问速度。Memcached 基于一个存储 key/value 对的 HashMap。Memcached 和 Redis 的应用场景很相似，区别是：① Redis 支持更多的数据种类；② Redis 支持主从数据备份；③ Redis 支持数据持久化。两种内存型数据库的性能都非常优秀。目前 Redis 应用在数据量较小的应用上，性能更优。Memcached 用在动态系统中，如文件、图片缓存场景，可以减少数据库负载、提升性能。
- MongoDB 是一个分布式 NoSQL 数据库，处理 NoSQL 非关系型数据存储。它的数据非常松散，面向集合及易存储对象类型数据存储，支持大型对象（如视频）的存储。MongoDB 支持集合存储，数据被分组存储在数据集中，每个数据集在数据库中都有一个唯一的标识名，类似于 key/value 的模式。目前 MongoDB 的典型应用如：①网站实时数据处理，处理实时的插入、更新、查询，支持实时数据的复制和高伸缩性；②用于系统数据缓存，由它搭建的持久化缓存层可以避免下层数据源过载；③用于高伸缩场景。MongoDB 非常适合数十或数百台服务器组成的数据库。其缺点是不支持级联的跨文档查询。

## 2.2 MySQL 关系型数据库

MySQL 是一个关系型数据库管理系统，由瑞典 MySQL AB 公司开发，目前属于 Oracle 旗下产品。MySQL 是最流行的关系型数据库管理系统，在 Web 应用方面是最好的 RDBMS (Relational Database Management System, 关系型数据库管理系统) 应用软件之一。全球主流的互联网公司的系统基本上都使用 MySQL 数据库，可以看出其强大的功能和强劲的性能。MySQL 5.7 现在已经可以轻松达到 50 万 QPS (Queries Per Second, 每秒查询率) 的性能，并支持 NoSQL 接口，通过 NoSQL 接口可以达到 100 万 QPS。



MySQL 是一种关联数据库管理系统，关联数据库将数据保存在不同的表中，而不是将所有数据放在一个大仓库内，这样就增加了速度并提高了灵活性。

MySQL 所使用的 SQL 语言是用于访问数据库最常用的标准化语言。MySQL 软件采用了双授权政策，它分为社区版和商业版，由于其体积小、速度快、总体拥有成本低，尤其是开放源码这一特点，一般中小型网站的开发都选择 MySQL 作为网站数据库。其社区版的性能卓越，搭配 PHP 和 Apache 可组成良好的开发环境。

与其他的大型数据库（如 Oracle、DB2、SQL Server 等）相比，MySQL 自有它的不足之处，但是这丝毫也没有减少它受欢迎的程度。对于一般的个人使用者和中小型企业来说，MySQL 提供的功能已经绰绰有余，而且由于 MySQL 是开放源码软件，因此可以大大降低总体拥有成本。

Linux（作为操作系统）、Apache 或 Nginx（作为 Web 服务器）、MySQL（作为数据库）、PHP/Perl/Python（作为服务器端脚本解释器）这 4 种软件都是免费或开源软件（Free/Libre and Open Source Software, FLOSS），因此使用这 4 种软件不用花一分钱（除开人工成本）就可以建立一个稳定、免费的网站系统，被业界称为 LAMP 或 LNMP 组合。

MySQL 支持单、复制、集群模式，分别适合小规模应用、中小规模应用和大规模应用，其应用场景非常广泛，在互联网系统中得到大量使用。基于 Nginx 的系统也遵循这样的原则和架构，可以组成低成本的系统。

MySQL 用于存储关系型数据，和 Redis、Memcached、MongoDB 等组成高并发、分布式的大型系统。互联网的特点是使用不同特点的产品解决不同问题，对外提供高速响应、大并发的系统，所以合理地使用各种系统是最主要的方面。想要利用每种系统的特点而规避其缺点考验的是对技术和系统的熟练掌握与总体架构能力。

---

**注意：**关系型数据指的是二维表及其之间联系的一种数据类型，关系操作主要为查询、选择、投影、连接、并、交、差，以及增加、删除、修改、查询等普通操作。

---

非关系型数据及数据库指的是数据之间没有明确的范围和定义，不需要预定义表结构、模式。数据主要是 key/value 模式，数据模型简单，支持 key/value 数据、列簇存储的数据、key/value 结构的文档数据以及图结构等。非关系型 NoSQL 数据库对数据一致性要求不高，但灵活性好，性能好。

所以对于数据的选择和使用，是基于对数据的分析和建模区分的。例如，用户的个人信息等就是结构化数据，一般使用 MySQL 存储。而网页检索记录、用户操作记录等信息就是 NoSQL 数据，一般使用 NoSQL 数据存储和查询更合适。通常中大型系统中的数据都是区别对待的。

## 2.2.1 yum 安装方法

安装 MySQL 有两种方法：一种是通过源码自行编译安装，这种方式适合高级用户定制

MySQL 特性，这里不做说明；另一种是通过编译过的二进制文件进行安装。二进制文件安装的方法又分为两种：一种是不针对特定平台的通用安装方法，使用的二进制文件是扩展名为 .tar.gz 的压缩文件；另一种是使用 RPM 或其他包进行安装，这种安装进程会自动完成系统相关配置，所以比较方便。

MySQL 运行需要很多依赖库，在安装过程中需要检查并逐一首先安装各种依赖库，过程比较复杂，所以在服务器联网的情况下，推荐使用 yum 安装。yum 基于 RPM 包管理，能够从指定的服务器自动下载 RPM 包并且安装，可以自动处理依赖性关系，并且一次安装所有依赖的软件包。利用 yum，这些检查和依赖库安装都会自动完成。只有当服务器不可访问时才使用手工安装。

本文重点不是 MySQL 的安装和管理，所以只介绍 yum 安装方法，其他安装方法如有需要，请读者自行了解。

---

**注意：**yum 是一个在 Fedora 和 RedHat 以及 CentOS 中的 Shell 前端软件包管理器。基于 RPM 包管理，能够从指定的服务器自动下载 RPM 包并且安装，可以自动处理依赖性关系，并且一次安装所有依赖的软件包，无须烦琐地一次次下载、安装。

---

可供 yum 下载的软件包包括 Fedora 本身的软件包以及源自 RPM Fusion 和 RPM 的 FedoraExtras，全部是由 Linux 社区维护的，并且基本是自由软件。所有的包都有一个独立的 GPG 签名，主要是为了系统安全。而对于 Fedora Core 4.0 的用户，RPM 的签名是自动导入并安装的。

yum 在 CentOS 上是默认安装的。例如，使用下面的 yum 命令可列出可安装的 MySQL 库：

```
yum list installed | grep mysql
```

### 1. 确认是否已经安装 MySQL

查看自带 MySQL 是否已安装。不同的 CentOS 安装模式可能已经安装了集成的 MySQL，使用下面命令检查其是否存在，如果存在则直接使用。

```
yum list installed | grep mysql
```

### 2. 卸载自带 MySQL

若有自带安装的 MySQL，需要卸载 CentOS 系统自带 MySQL 数据库，使用下面命令卸载：

```
yum -y remove mysql-libs.x86_64
```

若有多个依赖文件，则依次卸载。

### 3. 查看 yum 库上的 MySQL 版本信息

需要查看 yum 库上的 MySQL 版本信息时，使用命令：

```
yum list | grep mysql
```



或

```
yum -y list mysql*
```

#### 4. 使用 yum 安装 MySQL 数据库

当环境已经检查好，可以安装新的 MySQL 时，使用命令：

```
yum -y install mysql-server mysql mysql-devel
```

上面命令将 mysql-server、mysql、mysql-devel 都安装好，当结果显示为“Complete !”时，即安装完毕。

---

**注意：**安装 mysql 只是安装了数据库，只有安装 mysql-server 才相当于安装了客户端。

---

#### 5. 查看 MySQL 数据库版本信息

查看新安装 MySQL 数据库版本信息时使用命令：

```
rpm -qi mysql-server
```

---

**注意：**使用 root 安装 MySQL。

---

安装完成后，安装进程会在 Linux 中添加一个 mysql 组，以及属于 mysql 组的用户 mysql。可通过 id 命令查看：

```
id mysql
uid=496(mysql)gid=493(mysql) groups=493(mysql)
```

MySQL 服务器安装之后虽然配置了相关文件，但并没有自动启动 mysqld 服务，需要自行启动：

```
service mysqld start
Starting MySQL.. SUCCESS!
```

可以通过检查端口是否开启查看 MySQL 是否正常启动：

```
netstat -anp|grep 3306
tcp        0    0    0.0.0.0:3306    0.0.0.0:*    LISTEN    34693/mysqld
```

### 2.2.2 使用 mysql 测试服务

运行客户端程序 mysql，在 mysql/bin 目录中，测试能否连接到 mysqld。如果安装成功则应该可以运行 mysql 命令。注意，mysqld 服务必须已经开启。

```
mysql
Welcome to the MySQLmonitor.  Commands end with ; or \g.
Your MySQL connection id is 1
Server version: 5.7.16MySQL Community Server (GPL)
Copyright (c) 2000, 2016,Oracle and/or its affiliates. All rights reserved.
Oracle is a registered trademark of Oracle Corporation and/or its affiliates.
Other names may be trademarks of their respective owners.
```

Type 'help;' or '\h' for help. Type '\c' to clear the current input statement.  
mysql>

对于 yum 安装, 安装程序已经把可执行程序放到系统程序目录里, 不需要修改环境变量, 可以直接使用。

### 2.2.3 MySQL 文件分布

以 yum 方式安装后的 MySQL 文件分布如表 2-1 所示。

表 2-1 MySQL 文件分布

目录	内容
/usr/bin	客户端程序和脚本
/usr/sbin	程序进程 mysqld
/var/lib/mysql	日志文件和数据库文件
/usr/share/info	手册
/usr/share/man	UNIX 联机手册
/usr/include/mysql	头文件
/usr/lib/mysql	库文件
/usr/share/mysql	支撑类文件, 包括错误码、字符集、示例配置文件、SQL 安装脚本等

### 2.2.4 数据库操作

除去安装部分提到的 mysql 命令行工具, 推荐在 Windows 上使用 Navicat 工具进行可视化管理和调试。方便的工具可以提高工作效率。这里简单介绍一下 Navicat 工具。

Navicat 是一个图形化的工具, 下载并安装 Navicat, 运行后主界面如图 2-2 所示。

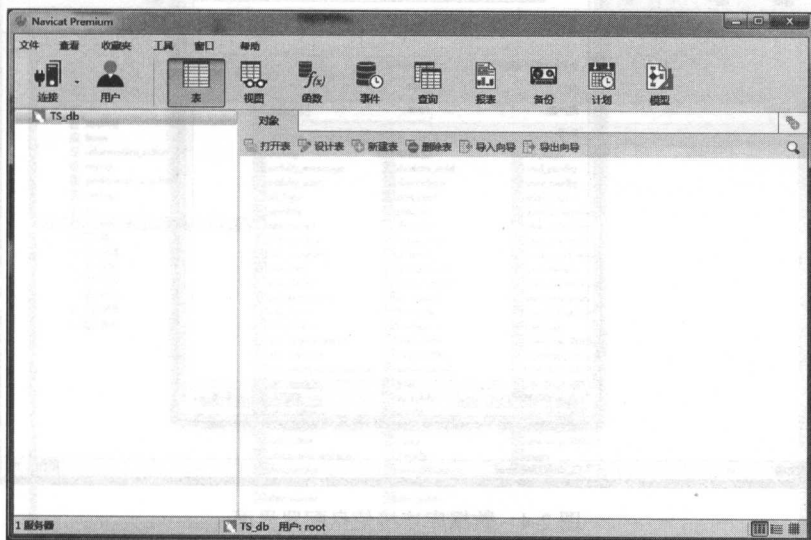


图 2-2 Navicat 运行主界面

首先需要创建一个连接，单击“连接”按钮，弹出选择数据库类型菜单，如图 2-3 所示。

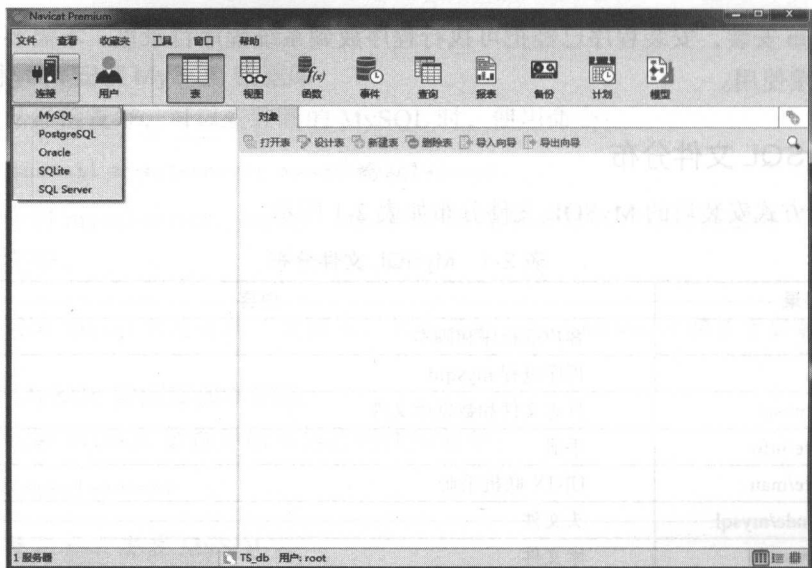


图 2-3 选择数据库类型

这里有 5 种常用数据库，在这里我们选择 MySQL，然后弹出数据库连接信息配置界面，如图 2-4 所示。

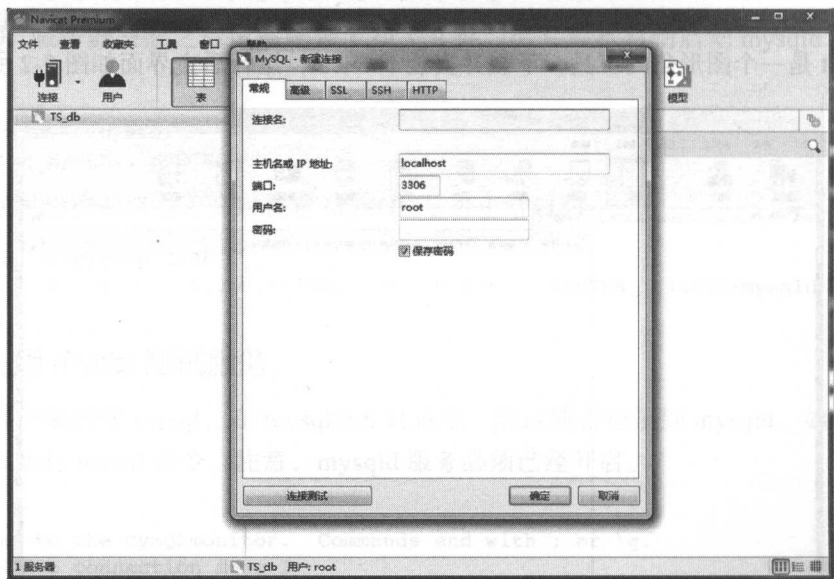


图 2-4 数据库连接信息配置界面

根据数据库配置信息，输入连接名、地址、端口（一般是默认端口）、用户名、密码，

就可以连接数据库了。如果要检测数据是否正确，可以使用“连接测试”功能进行测试。

设置完成后，左侧列表将列出已经配置的数据库，双击即可连接到数据库，如图 2-5 所示。

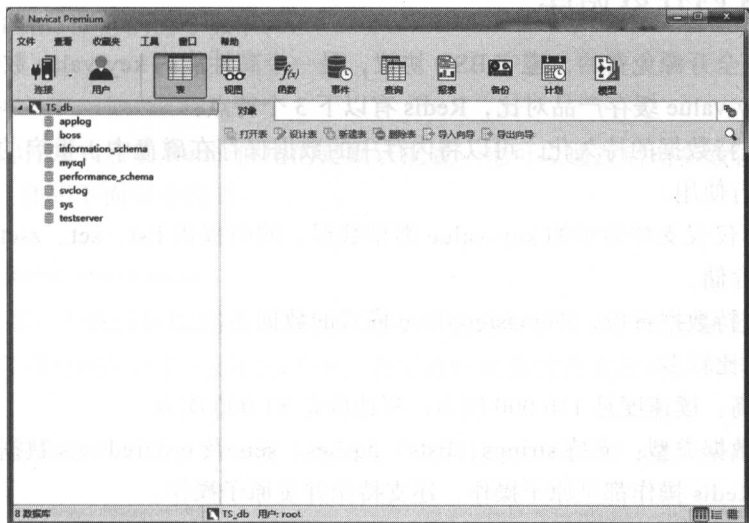


图 2-5 已经配置的数据库

对应连接上的所有数据库都会列出来，然后我们可以在上面执行各种操作。例如，新的数据库实现是空的，可以通过执行 SQL 脚本建立数据库，或使用图形化的界面手工建立数据库。例如，我们使用 testserver 库，单击 testserver 后，Navicat 将列出这个库中的资源，如图 2-6 所示。

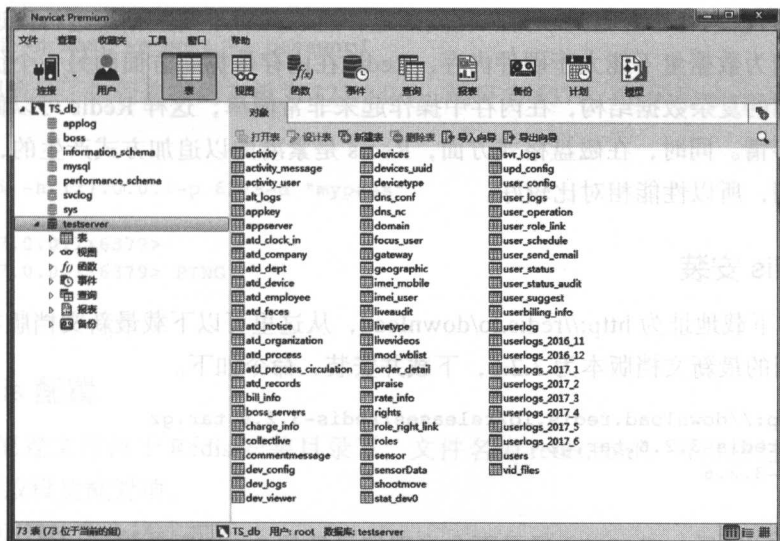


图 2-6 testserver 库资源

因为操作是全图形化的，所以具体的操作是比较容易理解和学习的，请读者自行研究。

## 2.3 Redis 内存数据库

Redis 是完全开源免费的，遵守 BSD 协议，是一个高性能的 key-value 数据库。

与其他 key-value 缓存产品对比，Redis 有以下 3 个特点：

- Redis 支持数据的持久化，可以将内存中的数据保存在磁盘中，重启的时候可以再次加载进行使用。
- Redis 不仅仅支持简单的 key-value 类型数据，同时提供 list、set、zset、hash 等数据结构的存储。
- Redis 支持数据备份，即 master-slave 模式的数据备份。

Redis 优势比较多：

- 性能极高：读速度是 110 000 次 /s，写速度是 81 000 次 /s。
- 丰富的数据类型：支持 strings、lists、hashes、sets 及 ordered sets 数据类型。
- 原子：Redis 操作都是原子操作，还支持全并发原子操作。
- 丰富的特性：支持 publish/subscribe、通知、key 过期等特性。
- 持久化：支持以 RDB 和 AOF 方式将数据持久化到文件中。
- 数据复制：Redis 的主从复制功能非常强大，一个 master 可以拥有多个 slave，而一个 slave 又可以拥有多个 slave，如此下去，形成了强大的多级服务器集群架构。

Redis 有着更复杂的数据结构并且提供原子操作，不同于其他数据库，Redis 的数据类型都是基于基本数据结构的，无须进行额外的抽象。

Redis 运行在内存中并且可以持久化到磁盘，所以在对不同数据集进行高速读写时需要权衡内存，因为数据量不能大于硬件内存。Redis 在内存数据库方面的另一个优点是，相比在磁盘上相同的复杂数据结构，在内存中操作起来非常简单，这样 Redis 可以做很多内部复杂性很强的事情。同时，在磁盘格式方面，Redis 是紧凑的以追加方式产生的，它并不需要进行随机访问，所以性能相对比较好。

### 2.3.1 Redis 安装

Redis 的下载地址为 <http://redis.io/download>，从这里可以下载最新文档版本。

本书使用的最新文档版本为 3.2.6，下载并安装，命令如下。

```
wget http://download.redis.io/releases/redis-3.2.6.tar.gz
tar xzf redis-3.2.6.tar.gz
cd redis-3.2.6
make
```

安装完 Redis 后 redis-3.2.6 目录下会出现编译后的 Redis 服务程序 redis-serve，还有用

于测试的客户端程序 `redis-cli`，两个程序位于源码目录 `src` 下。

### 2.3.2 启动 Redis 服务

默认启动 Redis 服务的命令如下：

```
cd src
./redis-server
```

注意，以这种方式启动 Redis 使用的是默认配置。也可以通过启动参数告诉 Redis 使用指定配置文件，使用下面命令启动：

```
cd src
./redis-server redis.conf
```

`redis.conf` 是一个默认的配置文。我们可以根据需要使用自己的配置文件。`redis.conf` 的配置参数和选项直接注释在 `redis.conf` 中，需要的时候直接查看注释，然后配置相关选项就可以使用了。

启动 Redis 服务进程后，就可以使用测试客户端程序 `redis-cli` 和 `redis-server` 交互。例如：

```
cd src
./redis-cli

redis>set foo bar
OK
redis>get foo
"bar"
```

如果需要在远程 Redis 服务上执行命令，使用的也是 `redis-cli` 命令。

语法如下：

```
redis-cli -h host -p port -a password
```

下面实例演示了如何连接到主机为 `127.0.0.1`、端口为 `6379`、密码为 `mypass` 的 Redis 服务上。

```
redis-cli -h 127.0.0.1 -p 6379 -a "mypass"

redis 127.0.0.1:6379>
redis 127.0.0.1:6379> PING

PONG
```

### 2.3.3 Redis 配置

Redis 的配置文件位于 Redis 安装目录下，文件名为 `redis.conf`。可以通过 `CONFIG` 命令查看配置项或设置配置项。

Redis `CONFIG` 命令格式如下：

```
redis 127.0.0.1:6379> CONFIG GET CONFIG_SETTING_NAME
```

示例:

```
redis 127.0.0.1:6379> CONFIG GET loglevel
1)"loglevel"
2)"notice"
```

可以使用 \* 号获取所有配置项:

```
redis 127.0.0.1:6379> CONFIG GET *
```

```
1)"dbfilename"
2)"dump.rdb"
3)"requirepass"
4)""
5)"masterauth"
6)""
7)"unixsocket"
8)""
9)"logfile"
10)""
11)"pidfile"
12)""
13)"slave-announce-ip"
14)""
15)"maxmemory"
16)"0"
17)"maxmemory-samples"
18)"5"
19)"timeout"
20)"0"
21)"auto-aof-rewrite-percentage"
22)"100"
23)"auto-aof-rewrite-min-size"
24)"67108864"
25)"hash-max-ziplist-entries"
26)"512"
27)"hash-max-ziplist-value"
28)"64"
29)"list-max-ziplist-size"
30)"-2"
31)"list-compress-depth"
32)"0"
33)"set-max-intset-entries"
34)"512"
35)"zset-max-ziplist-entries"
36)"128"
37)"zset-max-ziplist-value"
38)"64"
39)"hll-sparse-max-bytes"
40)"3000"
41)"lua-time-limit"
42)"5000"
43)"slowlog-log-slower-than"
44)"10000"
45)"latency-monitor-threshold"
46)"0"
```



```

47)"slowlog-max-len"
48)"128"
49)"port"
50)"6379"
51)"tcp-backlog"
52)"511"
53)"databases"
54)"16"
55)"repl-ping-slave-period"
56)"10"
57)"repl-timeout"
58)"60"
59)"repl-backlog-size"
60)"1048576"
61)"repl-backlog-ttl"
62)"3600"
63)"maxclients"
64)"10000"
65)"watchdog-period"
66)"0"
67)"slave-priority"
68)"100"
69)"slave-announce-port"
70)"0"
71)"min-slaves-to-write"
72)"0"
73)"min-slaves-max-lag"
74)"10"
75)"hz"
76)"10"
77)"cluster-node-timeout"
78)"15000"
79)"cluster-migration-barrier"
80)"1"
81)"cluster-slave-validity-factor"
82)"10"
83)"repl-diskless-sync-delay"
84)"5"
85)"tcp-keepalive"
86)"300"
87)"cluster-require-full-coverage"
88)"yes"
89)"no-appendfsync-on-rewrite"
90)"no"
91)"slave-serve-stale-data"
92)"yes"
93)"slave-read-only"
94)"yes"
95)"stop-writes-on-bgsave-error"
96)"yes"
97)"daemonize"
98)"no"
99)"rdbcompression"
100)"yes"
101)"rdbchecksum"

```

```

102)"yes"
103)"activerehashing"
104)"yes"
105)"protected-mode"
106)"yes"
107)"repl-disable-tcp-nodelay"
108)"no"
109)"repl-diskless-sync"
110)"no"
111)"aof-rewrite-incremental-fsync"
112)"yes"
113)"aof-load-truncated"
114)"yes"
115)"maxmemory-policy"
116)"noeviction"
117)"loglevel"
118)"notice"
119)"supervised"
120)"no"
121)"appendfsync"
122)"everysec"
123)"syslog-facility"
124)"local0"
125)"appendonly"
126)"no"
127)"dir"
128)"/root/redis-3.2.6/src"
129)"save"
130)"3600 1 300 100 60 10000"
131)"client-output-buffer-limit"
132)"normal 0 0 0 slave 268435456 67108864 60 pubsub 33554432 8388608 60"
133)"unixsocketperm"
134)"0"
135)"slaveof"
136)""
137)"notify-keyspace-events"
138)""
139)"bind"
140)""

```

### 2.3.4 参数说明

Redis 配置除了通过命令配置外，还可以通过直接编译 `redis.conf` 进行修改。默认的 `redis.conf` 对每一个配置项提供了详细的英文说明，需要使用的时候参照说明进行配置即可，本节主要介绍常用的配置项，方便学习时使用。

#### (1) `daemonize no`

Redis 默认不以守护进程的方式运行，可以通过该配置项修改，使用 `yes` 启用守护进程 (`daemonize yes`)。

#### (2) `pidfile /var/run/redis.pid`

当 Redis 以守护进程方式运行时，Redis 默认会把 `pid` 写入 `/var/run/redis.pid` 文件，可以

通过 pidfile 指定。

(3) port 6379

该配置项用于指定 Redis 监听端口，默认端口为 6379。

(4) bind 127.0.0.1

该配置项用于绑定主机地址。

(5) timeout 300

该配置项用于指定当客户端闲置多长时间后关闭连接，如果指定为 0，表示关闭该功能。

(6) loglevel verbose

该配置项用于指定日志记录级别，Redis 支持 4 个级别：debug、verbose、notice、warning，默认为 verbose。

(7) logfile stdout

该配置项用于指定日志记录方式，默认为标准输出，如果配置 Redis 为守护进程方式运行，而这里又配置为日志记录方式为标准输出，则日志将会发送给 /dev/null。

(8) databases 16

该配置项用于设置数据库的数量，默认数据库为 0，可以使用 SELECT <dbid> 命令在连接上指定数据库 ID。

(9) save<seconds><changes>

该配置项用于指定在多长时间內，有多少次更新操作，就将数据同步到数据文件，可以多个条件配合。

Redis 默认配置文件中提供了 3 个条件：

```
save 900 1
save 300 10
save 60 10000
```

这 3 个条件分别表示 900 秒（15 分钟）内有 1 个更改，300 秒（5 分钟）内有 10 个更改，以及 60 秒内有 10000 个更改。

(10) rdbcompression yes

该配置项用于指定存储至本地数据库时是否压缩数据，默认为 yes。Redis 采用 LZF 压缩，如果为了节省 CPU 时间，可以关闭该选项，但会导致数据库文件变得巨大。

(11) dbfilename dump.rdb

该配置项用于指定本地数据库文件名，默认值为 dump.rdb。

(12) dir ./

该配置项用于指定本地数据库存放目录。

(13) slaveof<masterip><masterport>

该配置项用于当本机为 slave 服务时，设置 master 服务的 IP 地址及端口，在 Redis 启动时，它会自动从 master 进行数据同步。

## (14) masterauth&lt;master-password&gt;

当 master 服务设置了密码保护时, slave 服务连接 master 的密码。

## (15) requirepass foobared

该配置项用于设置 Redis 连接密码, 如果配置了连接密码, 客户端在连接 Redis 时需要通过 AUTH <password> 命令提供密码, 默认关闭。

## (16) maxclients 128

该配置项用于设置同一时间最大客户端连接数, 默认无限制。Redis 可以同时打开的客户端连接数为 Redis 进程可以打开的最大文件描述符数量, 如果设置 maxclients 0, 表示无限制。当客户端连接数到达限制时, Redis 会关闭新的连接并向客户端返回 max number of clients reached 错误信息。

## (17) maxmemory&lt;bytes&gt;

该配置项用于指定 Redis 最大内存限制, Redis 在启动时会把数据加载到内存中, 达到最大内存后, Redis 会先尝试清除已到期或即将到期的 key, 若当此方法处理后仍然到达最大内存设置, 则将无法再进行写入操作, 但仍然可以进行读取操作。Redis 新的 VM 机制会把 key 存放在内存, value 存放在 swap 区。

## (18) appendonly no

该配置项用于指定是否在每次更新操作后进行日志记录, 在默认情况下, Redis 异步地把数据写入磁盘, 如果不开启, 可能会在断电时导致一段时间内的数据丢失。因为 Redis 本身同步数据文件是按 save 条件来同步的, 所以有的数据会在一段时间内只存在于内存中。默认为 no。

## (19) appendfilename appendonly.aof

该配置项用于指定更新日志文件名, 默认为 appendonly.aof。

## (20) appendfsync everysec

该配置项用于指定更新日志条件, 共有 3 个可选值。

1) no: 表示等操作系统进行数据缓存同步到磁盘(快)。

2) always: 表示每次更新操作后手动调用 fsync() 将数据写到磁盘(慢, 安全)。

3) everysec: 表示每秒同步一次(折中, 默认值)。

## (21) vm-enabled no

该配置项用于指定是否启用虚拟内存机制, 默认值为 no。简单地介绍一下, VM 机制将数据分页存放, 由 Redis 将访问量较少的页即冷数据存放到磁盘上, 访问多的页面由磁盘自动换出到内存中。

## (22) vm-swap-file /tmp/redis.swap

该配置项用于指定虚拟内存文件路径, 默认值为 /tmp/redis.swap, 不可多个 Redis 实例共享。

## (23) vm-max-memory 0

将所有大于 `vm-max-memory` 的数据存入虚拟内存，无论 `vm-max-memory` 设置多小，所有索引数据都是内存存储的（Redis 的索引数据就是 `keys`），也就是说，当 `vm-max-memory` 设置为 0 时，其实是所有 `value` 都存在于磁盘中。默认值为 0。

#### (24) `vm-page-size 32`

Redis `swap` 文件分成了很多的 `page`，一个对象可以保存在多个 `page` 中，但一个 `page` 不能被多个对象共享，`vm-page-size` 是要根据存储的数据大小来设定的：如果存储很多小对象，`page` 大小最好设置为 32B 或者 64B；如果存储很大对象，则可以使用更大的 `page`，如果不确定，就使用默认值。

#### (25) `vm-pages 134217728`

该配置项用于设置 `swap` 文件中的 `page` 数量，由于 `page` 表（一种表示页面空闲或使用的 `bitmap`）是放在内存中的，在磁盘上每 8 个 `pages` 将消耗 1B 的内存。

#### (26) `vm-max-threads 4`

该配置项用于设置访问 `swap` 文件的线程数，最好不要超过机器的核数，如果设置为 0，那么所有对 `swap` 文件的操作都是串行的，可能会造成比较长时间的延迟。默认值为 4。

#### (27) `glueoutputbuf yes`

该配置项用于设置在向客户端应答时，是否把较小的包合并为一个包发送，默认为开启。

#### (28) `hash-max-zipmap-entries 64`、`hash-max-zipmap-value 512`

该配置项用于指定在超过一定的数量或者最大的元素超过某一临界值时，采用一种特殊的哈希算法。

#### (29) `activeresharding yes`

该配置项用于指定是否激活重置哈希，默认为开启。

#### (30) `include /path/to/local.conf`

该配置项用于指定包含其他的配置文件，可以在同一主机上多个 Redis 实例之间使用同一份配置文件，而各个实例又同时拥有自己特定的配置文件。

## 2.3.5 数据类型

Redis 支持 5 种数据类型：`string`（字符串）、`hash`（哈希）、`list`（列表）、`set`（集合）及 `zset`（sorted set，有序集合）。

### 1. `string`（字符串）

`string` 是 Redis 最基本的数据类型，可以理解成与 `memcached` 一样的类型，一个 `key` 对应一个 `value`，一个 `key` 的最大存储容量为 512MB。`string` 类型是二进制安全的，即 Redis 的 `string` 可以包含任何数据。如 JPG 图片或者序列化的对象。

例如：

```
redis 127.0.0.1:6379> SET name "test"
OK
redis 127.0.0.1:6379> GET name"test"
```

在以上实例中，我们使用了 Redis 的 SET 和 GET 命令，键（key）为 name，对应的值（value）为 test。

## 2. hash（哈希）

Redis hash 是一个 key-value 对集合。Redis hash 是一个 string 类型的 field 和 value 的映射表，特别适合用于存储对象。

例如：

```
127.0.0.1:6379> HMSET user:1 username test password test points 200
OK
127.0.0.1:6379> HGETALL user:1
1)"username"
2)"test"
3)"password"
4)"test"
5)"points"
6)"200"
127.0.0.1: 6379>
```

以上实例中，hash 数据类型存储了包含用户脚本信息的用户对象。实例使用了 Redis HMSET、HGETALL 命令，user:1 为 key 值。

每个 hash 可以存储  $2^{32}-1$  个 key-value 对（40 多亿）。

## 3. list（列表）

Redis 列表是简单的字符串列表，按照插入顺序排序。你可以添加一个元素到列表的头部（左边）或者尾部（右边）。

例如：

```
redis 127.0.0.1:6379> lpush test redis(integer)1
redis 127.0.0.1:6379> lpush test mongodb(integer)2
redis 127.0.0.1:6379> lpush test rabbitmq(integer)3
redis 127.0.0.1:6379> lrange test 0 10
1)"rabbitmq"
2)"mongodb"
3)"redis"
redis 127.0.0.1:6379>lrange test 5 10
(empty list or set)
```

列表最多可存储  $2^{32}-1$  个元素（40 多亿）。

## 4. set（集合）

Redis 的 set 是 string 类型的无序集合。集合是通过哈希表实现的，所以添加、删除、查找的复杂度都是  $O(1)$ 。sadd 命令用于添加一个 string 元素到 key 对应的 set 集合中，成功则返回 1，如果元素已经在集合中则返回 0，key 对应的 set 不存在则返回错误。

例如：

```
redis 127.0.0.1:6379> sadd test redis(integer)1
redis 127.0.0.1:6379> sadd test mongodb(integer)1
redis 127.0.0.1:6379> sadd test rabbitmq(integer)1
redis 127.0.0.1:6379> sadd test rabbitmq(integer)0
redis 127.0.0.1:6379> smembers test
1)"rabbitmq(integer)1"
2)"mongodb(integer)1"
3)"redis(integer)1"
```

---

**注意：**在以上实例中，rabbitmq 添加了两次，但根据集合内元素的唯一性，第二次插入的元素将被忽略。

---

集合中最大的成员数为  $2^{32}-1$  (40 多亿)。

### 5. zset (有序集合)

Redis zset 也是 string 类型元素的集合，且不允许重复的成员。不同的是，zset 的每个元素都会关联一个 double 类型的分数。Redis 正是通过分数为集合中的成员进行从小到大排序的。zset 的成员是唯一的，但分数 (score) 可以重复。

例如：

```
redis 127.0.0.1:6379> zadd test 0 redis(integer)1
redis 127.0.0.1:6379> zadd test 0 mongodb(integer)1
redis 127.0.0.1:6379> zadd test 0 rabbitmq(integer)1
redis 127.0.0.1:6379> zadd test 0 rabbitmq(integer)0
redis 127.0.0.1:6379> ZRANGEBYSCORE test 0 1000
1)"mongodb(integer)1"
2)"rabbitmq(integer)0"
3)"rabbitmq(integer)1"
4)"redis(integer)1"
```

## 2.4 PostgreSQL 关系型数据库

PostgreSQL 是以加州大学伯克利分校计算机系开发的 POSTGRES (现在已经更名为 PostgreSQL) 4.2 版本为基础开发的对象关系型数据库管理系统 (ORDBMS)。它是一个自由的对象关系型数据库服务器 (数据库管理系统)，在灵活的 BSD 风格许可证下发行。PostgreSQL 提供了相对其他开源数据库系统 (如 MySQL 和 Firebird) 和专有系统 (如 Oracle、Sybase、DB2 和 Microsoft SQL Server) 之外的另一种选择。

PostgreSQL 支持大部分 SQL 标准并且提供了许多其他现代特性：复杂查询、外键、触发器、视图、事务完整性、MVCC。同样，PostgreSQL 可以用许多方法扩展，例如，增加新的数据类型、函数、操作符、聚集函数、索引。PostgreSQL 的特性覆盖了 SQL-2/SQL-92 和 SQL-3/SQL-99，首先，它包括了目前世界上最丰富的数据类型支持。

从技术角度来讲，PostgreSQL 采用的是比较经典的 C/S (Client/Server) 结构，也就是



一个客户端对应一个服务器端守护进程的模式，这个守护进程分析从客户端发来的查询请求，生成规划树，进行数据检索并最终把结果格式化输出后返回给客户端。为了便于客户端程序编写，由数据库服务器提供统一的客户端 C 接口。不同的客户端接口都是源自这个 C 接口，如 ODBC、JDBC、Python、Perl、C/C++、ESQL 等。PostgreSQL 支持的接口非常丰富，几乎支持所有类型数据库客户端接口。

要在 CentOS 6.4 上使用 yum，首先要保证 CentOS 外网是连通的，如果外网不通，只能使用 RPM 包或源码的方法安装。本文主要研究在 Lua 中如何使用 PostgreSQL，所以以 yum 简化安装过程为例。

这里使用 PostgreSQL yum repository 安装最新版本的 PostgreSQL。

### 1. 安装 PostgreSQL yum repository

yum 默认安装的 PostgreSQL 是 8.X 版本的，如果想体验新版本 PostgreSQL 的性能，需要安装 9.X 版本，这需要首先安装 PostgreSQL yum repository。如果只想调试 OpenResty 的 PG 访问代码，可以跳过本步，直接使用 yum 安装 8.X 版本的 PG。

使用下面命令安装 PostgreSQL yum repository：

```
yum -i http://download.postgresql.org/pub/repos/yum/9.6/redhat/
rhel-6.4-x86_64/pgdg-redhat96-9.6-3.noarch.rpm
```

只要命令正确，就可以看到安装成功。如果要使用最新版本的 PG，可以在 <http://yum.postgresql.org> 上查找到对应 RPM 包的连接，然后执行 `rpm -i uri` 命令。

### 2. 安装 PostgreSQL

使用下面命令安装新版本 PG：

```
yum install postgresql92-server postgresql92-contrib
```

如果没有安装 PostgreSQL yum repository，使用下面的命令安装 8.X 版本：

```
yum install postgresql*-server postgresql*-contrib
```

### 3. 查看安装

使用 rpm 命令查看是否安装成功：

```
rpm -qa | grep postgresql
```

如果看到了类似：

```
postgresql96-server-9.6.1-1PGDG.rhel6.x86_64
postgresql96-libs-9.6.1-1PGDG.rhel6.x86_64
postgresql96-9.6.1-PGDG.rhel6.x86-64
postgresql96-contrib-9.6.1-1PGDG.rhel6.x86_64
```

表示安装成功，否则根据错误提示修正错误后重新安装。

### 4. 初始化并启动数据库

PG 安装完成需要首先初始化：

```
service postgresql-9.6 initdb
```

看到 Initializing database: [OK], 就可以启动数据库了:

```
service postgresql-9.6 start
```

看到 OK 字样, 表示 PostgreSQL 启动成功。

## 5. 测试

首先切换到 postgres 用户:

```
su - postgres
```

执行 psql 命令查看数据库列表:

```
psql -l
```

看到下面的 3 个数据库列表, 表示数据库启动并初始化成功:

```
List of databases
```

Name	owner	Encoding	Collate	Ctype	Access privileges
postgres	postgres	UTF8	en_us.UTF-8	en_us.UTF-8	
template0	postgres	UTF8	en_us.UTF-8	en_us.UTF-8	
template1	postgres	UTF8	en_us.UTF-8	en_us.UTF-8	

(3 rows)

---

**注意:** 因为 PostgreSQL 是一个关系型数据库, 所以管理和使用相对比较复杂, 推荐使用 Navicat 进行图形化的操作。

---

## 2.5 Memcached 内存数据库

Memcached 是一个高性能分布式内存对象缓存系统, 用于动态 Web 应用以减轻数据库负载。它通过在内存中缓存数据和对象来减少读取数据库的次数, 从而提高数据库驱动网站的速度。Memcached 基于一个存储键 / 值对的 HashMap。

Memcached 是一套分布式快取系统, 当初是 Danga Interactive 为了 LiveJournal 所发展的, 但被许多软件 (如 MediaWiki) 所使用。这是一套开源软件, 以 BSD license 授权协议发布。

Memcached 的 API 使用 32 位循环冗余校验 (CRC-32) 计算键值后, 将资料分散在不同的机器上。当表格满了以后, 接下来新增的资料会以 LRU 机制替换。由于 Memcached 通常只是当作快取系统使用, 所以使用 Memcached 的应用程序在写回较慢的系统时 (像是后端的数据库) 需要额外的程序码更新 Memcached 内的资料。

Memcached 是以 LiveJournal 旗下 Danga Interactive 公司的 Brad Fitzpatrick 为首开发的一款软件, 已成为 Mixi、Hatena、Facebook、Vox、LiveJournal 等众多服务中提高 Web 应用扩展性的重要因素。许多 Web 应用都将数据保存到 RDBMS 中, 应用服务器从中读取数

据并在浏览器中显示。但随着数据量的增大、访问的集中,就会出现 RDBMS 的负担加重、数据库响应恶化、网站显示延迟等重大影响。Memcached 的出现有效解决了以上这些问题。

Memcached 是高性能的分布式内存缓存服务器,一般的使用目的是通过缓存数据库查询结果,减少数据库访问次数,从而提高动态 Web 应用的速度,提高可扩展性。

Memcached 的守护进程 (daemon) 是用 C 编写的,但是客户端可以用任何语言来编写,并通过 Memcached 协议与守护进程通信。但是它并不提供冗余 (例如,复制其 HashMap 条目),当某个服务器 S 停止运行或崩溃了,所有存放在 S 上的键 / 值对都将丢失。

Memcached 由 Danga Interactive 开发,其最新版本发布于 2010 年 (作者为 Anatoly Vorobey 和 Brad Fitzpatrick),用于提升 LiveJournal.com 访问速度。LiveJournal (LJ) 每秒动态页面访问量达几千次,用户 700 万。Memcached 将数据库负载大幅度降低,更好地分配资源,用户可以更快速地访问。

为了提高性能,Memcached 中保存的数据都存储在 Memcached 内置的内存存储空间中。由于数据仅存在于内存中,因此重启 Memcached、重启操作系统会导致全部数据消失。另外,内存容量达到指定值之后,就基于 LRU (Least Recently Used) 算法自动删除不使用的缓存。Memcached 本身是为缓存而设计的服务器,因此并没有过多考虑数据的永久性问题。

尽管 Memcached 是“分布式”缓存服务器,但服务器端并没有分布式功能。各个 Memcached 不会互相通信以共享信息。那么,怎样进行分布式呢?这完全取决于客户端的实现。

### 2.5.1 Memcached 安装

本书使用 yum 安装 Memcached。Memcached 依赖于 libevent 库,这些依赖 yum 都会自动安装。

安装命令:

```
yum -y install memcached
```

启动和停止命令:

```
service memcached start|stop
```

Memcached 的配置是通过命令行进行的,如果只是学习 Memcached 的使用,使用 service 命令就可以了。

注意: 如果安装缺少其他支持,可以使用以下命令

```
yum groupinstall "Development Tools"
```

### 2.5.2 连接编辑

可以通过 telnet 命令并指定主机 IP 和端口来连接 Memcached 服务。

语法如下：

```
telnet HOST PORT
```

命令中的 HOST 和 PORT 为运行 Memcached 服务的 IP 和端口。

下面实例演示了如何连接到 Memcached 服务并执行简单的 set 和 get 命令。Memcached 默认端口为 11211。

```
telnet 127.0.0.1 11211
Trying 127.0.0.1...
Connected to 127.0.0.1.
Escape character is '^]'.
set foo 0 0 3           // 保存命令
bar                     // 数据
STORED                  // 结果
get foo                 // 取得命令
VALUE foo 0 3           // 数据
bar                     // 数据
END                     // 结束行
```

```
quit
```

Memcached 缺乏认证以及安全管制，因此应该将 Memcached 服务器放置在防火墙后。

### 2.5.3 管理 Memcached 服务

#### 1. 启动 Memcached

一般情况下，可以使用类似如下形式，启动 Memcached 服务：

```
memcached -d -m 64 -I 20m -u root -l 192.168.4.86 -p 11211 -c 1024 -P /tmp /
memcached.pid
```

上述命令行中，基于上面各个选项，以及其他一些选项的含义，表 2-2 列出了 Memcached 的常用选项。

表 2-2 Memcached 常用选项

选项	说明
-d	指定 Memcached 进程作为一个守护进程启动
-m <num>	指定分配给 Memcached 使用的内存，单位是 MB
-u <username>	运行 Memcached 的用户
-l <ip_addr>	监听的服务器 IP 地址，如果有多个地址，使用逗号分隔，格式可以为“IP 地址:端口号”，例如，-l 指定 192.168.0.184:19830, 192.168.0.195:13542；端口号也可以通过 -p 选项指定
-p <num>	Memcached 监听的端口，要保证该端口号未被占用
-c <num>	设置最大运行的并发连接数，默认是 1024
-R <num>	为避免客户端饿死，对连续达到的客户端请求数设置一个限额，如果超过该设置，会选择另一个连接来处理请求，默认为 20
-k	设置锁定所有分页的内存，对于大缓存应用场景，须谨慎使用该选项
-P	保存 Memcached 进程的 pid 文件

(续)

选项	说明
-s <file>	指定 Memcached 用于监听的 UNIX socket 文件
-a <perms>	设置 -s 选项指定的 UNIX socket 文件权限
-U <num>	指定监听 UDP 的端口，默认 11211，0 表示关闭
-M	当内存使用超出配置值时，禁止自动清除缓存中的数据项，此时 Memcached 不可用，直到内存被释放
-r	设置产生的 core 文件大小
-f <factor>	用于计算缓存数据项内存块大小的乘数因子，默认是 1.25
-n	为缓存数据项的 key、value、flag 设置最小分配字节数，默认是 48B
-C	禁用 CAS
-h	显示 Memcached 版本和摘要信息
-v	输出警告和错误信息
-vv	输出信息比 -v 更详细：不仅输出警告和错误信息，也输出客户端请求和响应信息
-i	输出 libevent 和 Memcached 的 licenses 信息
-t <threads>	指定用来处理请求的线程数，默认为 4
-D <char>	用于统计报告中 Key 前缀和 ID 之间的分隔符，默认是冒号 “:”
-L	尝试使用大内存分页 (pages)
-B <proto>	指定使用的协议，默认行为是自动协商 (autonegotiate)，可能使用的选项有 auto、ascii、binary
-I <size>	覆盖默认的 STAB 页大小，默认是 1MB
-F	禁用 flush_all 命令
-o <options>	指定逗号分隔的选项，一般用于扩展或实验性质的选项

-d 参数有如下几个选项，可以使用 -d 参数启动和关闭 Memcached 而不用使用 pid 文件。

- -d install 安装 Memcached 服务。
- -d uninstall：卸载 Memcached 服务。
- -d start：启动 Memcached 服务。
- -d restart：重启 Memcached 服务。
- -d stop：停止 Memcached 服务。
- -d shutdown：停止 Memcached 服务。

## 2. 停止 Memcached

可以在 Linux 下通过如下命令查询到 Memcached 进程号：

```
ps -ef | grep memcached
```

然后杀掉 Memcached 服务进程：

```
kill -9 <PID>
```

-9 表示强制杀掉进程，或

```
kill -9 `cat /tmp/memcached.pid`
```

也可以使用 -d 参数关闭 Memcached:

```
memcached -u root -d stop
```

## 2.5.4 Memcached 命令

本节介绍 Memcached 常用命令，方便读者后续使用。

### 1. stats 命令

该命令用于显示服务器信息、统计数据、结果示例数据等，例如：

```
stats
STAT pid 19510           // Memcache 进程 ID
STAT uptime 1466315      // 已运行秒数
STAT time 1339671194     // 服务器当前的 UNIX 时间戳
STAT version 1.4.4       // Memcache 版本号
.....
END
```

这里不再展开叙述每个选项的意义。

stats 命令有若干个二级子项，如表 2-3 所示。

表 2-3 stats 命令二级子项

命令	说明
stats slabs	显示各个 slab 的信息，包括 chunk 的大小、数目、使用情况等
stats items	显示各个 slab 中 item 的数目和最老的 item 年龄（最后一次访问距离现在的秒数）
stats detail	设置或者显示详细操作记录
[on off dump]	参数为 on，打开详细操作记录；参数为 off，关闭详细操作记录；参数为 dump，显示详细操作记录（每一个键值 get、set、hit、del 的次数）
stats malloc	打印内存分配信息
stats sizes	打印缓存使用信息
stats reset	重置统计信息

### 2. get 命令

get 命令用于获取缓存的数据，键为 key。语法格式：

```
get<key>*
```

示例：

```
get basis_behavior_user
```

结果如下所示：

```
VALUE basis_behavior_user 0
451{"aaData":[[{"d1a2233dc382432b8e19e40254fdb98a","100000002223484","1402563046319","c4f82195815300bcf39a5232707ad9c1","1","0","EBEST_W70","4.0.4","2.2.2","wifi","","EBEST","19","854*480","H-yun35","00:08:22:da:c1:ce","863531010517866","c4f82195815300bcf39a5232707ad9c11402562805664","460010255508963","1901589461","1402563045960","429","338","23197","1","0","0","2014-06-12_16:50:46","-1","0","3"]], "sEcho":1, "iTo
```

```
talRecords":0,"iTotalDisplayRecords":0}
END
```

也可以用 get 命令获取多个 key 对应的值, 如下所示:

```
get name hobby
VALUE name 1 7
1234567
VALUE hobby 0 25
tenis basketball football
END
```

### 3. set 命令

语法格式:

```
set<key><flags><exptime><bytes> [noreply]
<value>
```

noreplay 是可选参数, 告诉服务器不需要回复。

示例:

```
set name 0 1800 7
shirdrn
STORED
get name
VALUE name 0 7
shirdrn
END
```

### 4. delete 命令

delete 命令用于删除缓存中指定键 key 对应的数据。语法格式:

```
delete key [noreply]
```

参数说明:

key: 键值 key-value 结构中的 key, 用于查找缓存值。

noreply (可选): 告知服务器不需要返回数据。

示例:

```
set tValue 0 900 9
memcached
STORED
get tValue
VALUE tValue 0 9
memcached
END
delete tValue
DELETED
get tValue
END
delete tValue
NOT_FOUND
```



输出信息说明:

- DELETED: 删除成功。
- ERROR: 语法错误或删除失败。
- NOT\_FOUND: key 不存在。

## 5. add 命令

语法格式:

```
add<key><flags><exptime><bytes> [noreply]
<value>
```

示例:

```
add hobby 0 1800 10
basketball
STORED
get hobby
```

```
VALUE hobby 0 10
basketball
END
```

## 6. replace 命令

replace 命令用于覆盖一个已经存在的 key 及其对应 value, 替换时一定要保证替换后值的长度与原始长度相同, 否则覆盖失败。

语法格式:

```
replace<key><flags><exptime><bytes> [noreply]
<value>
```

示例:

```
get name
VALUE name 0 7
shirdrn
END
replace name 0 1800 7
youak47
STORED
get name
VALUE name 0 7
youak47
END
```

## 7. append 命令

append 命令用于在一个已经存在的数据值 (value) 上追加 (在数据值的后面追加)。

语法格式:

```
append<key><flags><exptime><bytes> [noreply]
<value>
```

示例:

```
get hobby
VALUE hobby 0 10
basketball
END
append hobby 0 1800 9
football
STORED
get hobby
VALUE hobby 0 19
basketball football
END
```

## 8. prepend 命令

prepend 命令用于在一个已经存在的数据值 (value) 上追加 (在数据值的前面追加)。

语法格式:

```
prepend<key><flags><exptime><bytes> [noreply]
<value>
```

示例:

```
get hobby
VALUE hobby 0 19
basketball football
END
prepend hobby 0 1800 6
tenis
STORED
get hobby
VALUE hobby 0 25
tenis basketball football
END
```

## 9. incr 命令

incr 命令是计数命令, 可以在原来已经存在的数字上进行累加求和计算, 并存储新的数值。

语法格式:

```
incr<key><value> [noreply]
```

示例:

```
set active_users 0 1000000 7
1000000
STORED
get active_users
VALUE active_users 0 7
1000000
END
```

```
incr active_users 99
1000099
```

### 10. decr 命令

decr 命令是计数命令，可以在原来已经存在的数字上进行减法计算，并存储新的数值。

语法格式：

```
decr<key><value> [noreply]
```

示例：

```
get active_users
VALUE active_users 0 7
1000099
END
decr active_users 3456
996643
```

### 11. flush\_all 命令

flush\_all 命令用于使缓存中的数据项失效，可选参数指定数据项在多少秒后失效。

语法格式：

```
flush_all [<time>] [noreply]
```

调用 flush\_all 命令时，数据所占的内存并不会被释放，但会被标记为过期，不能再被读取。后续添加的值会根据算法逐渐占用之前释放的空间。

示例：

```
setuser 0 0 5
123456
STORED
```

```
getuser
VALUE user0 5
123456
END
```

```
flush_all
OK
```

```
getuser
END
```

### 12. version 命令

version 命令用于返回 Memcached 服务器的版本信息。

示例：

```
version
VERSION 1.4.5
```

### 13. quit 命令

quit 命令用于退出 Telnet 终端。

## 2.6 MongoDB 分布式 NoSQL 数据库

MongoDB 是一个介于关系型数据库和非关系型数据库之间的产品，是非关系型数据库当中功能最丰富、最像关系型数据库的；它支持的数据结构非常松散，是类似 JSON 的 BSON 格式，因此可以存储比较复杂的数据类型。MongoDB 最大的特点是支持的查询语言非常强大，其语法类似于面向对象的查询语言，几乎可以实现类似关系型数据库单表查询的绝大部分功能，而且支持对数据建立索引。

MongoDB 的设计目标是高性能、可扩展、易部署、易使用，存储数据非常方便。其主要功能特性如下。

1) 面向集合存储，容易存储对象类型的数据。在 MongoDB 中，数据被分组存储在集合中，集合类似 RDBMS 中的表，一个集合可以存储无限多的文档。

2) 模式自由，采用无模式结构存储。在 MongoDB 的集合中存储的数据是无模式的文档。采用无模式存储数据是集合区别于 RDBMS 表的一个重要特征。

3) 支持完全索引，可以在任意属性上建立索引，包含内部对象。MongoDB 的索引和 RDBMS 的索引基本一样，可以在指定属性、内部对象上创建索引以提高查询的速度。除此之外，MongoDB 还提供创建基于地理空间的索引能力。

4) 支持查询。MongoDB 支持丰富的查询操作，支持 SQL 中的大部分查询。

5) 强大的聚合工具。MongoDB 除了提供丰富的查询功能外，还提供强大的聚合工具，如 count、group 等，支持使用 MapReduce 完成复杂的聚合任务。

6) 支持复制和数据恢复。MongoDB 支持主从复制机制，可以实现数据备份、故障恢复、读扩展等功能。而基于副本集的复制机制提供了自动故障恢复的功能，确保集群数据不会丢失。

7) 使用高效的二进制数据存储，包括大型对象（如视频）。MongoDB 支持以二进制格式存储数据，可以保存任何类型的数据对象。

8) 自动处理分片，以支持云计算层次的扩展。MongoDB 支持集群自动切分数据，对数据进行分片可以使集群存储更多的数据，实现更大的负载，也能保证存储的负载均衡。

9) 支持 Perl、PHP、Java、C#、JavaScript、Ruby、C 和 C++ 语言的驱动程序。MongoDB 提供了当前所有主流开发语言的数据库驱动包，开发人员使用任何一种主流开发语言都可以轻松编程，实现访问 MongoDB 数据库。

10) 文件存储格式为 BSON(JSON 的一种扩展)。BSON 是二进制格式的 JSON 的简称，BSON 支持文档和数组的嵌套。

11) 可以通过网络访问。可以通过网络远程访问 MongoDB 数据库。

“面向集合”(collection-oriented)，是指数据被分组存储在数据集中，这个数据集称为一个集合(collection)。每个集合在数据库中都有一个唯一的标识名，并且可以包含无限数目的文档。集合的概念类似关系型数据库(RDBMS)里的表(table)，不同的是它不需要定义

任何模式 (schema)。Nytro MegaRAID 技术中的闪存高速缓存算法,能够快速识别数据库内大数据集中的热数据,提供一致的性能改进。

模式自由 (schema-free),意味着对于存储在 MongoDB 数据库中的文件,我们不需要知道它的任何结构定义。

存储在集合中的文档,被存储为键-值 (key-value) 对的形式。键用于唯一标识一个文档,为字符串类型,而值可以是各种复杂的文件类型。我们称这种存储形式为 BSON (Binary Serialized Document Format)。

MongoDB 适合使用的场景如下:

1) 网站实时数据处理。MongoDB 非常适合实时的插入、更新与查询,并具备网站实时数据存储所需的复制及高度伸缩性。

2) 缓存。由于性能很高, MongoDB 适合作为信息基础设施的缓存层。在系统重启之后,由它搭建的持久化缓存层可以避免下层的数据源过载。

3) 高伸缩性的场景。MongoDB 非常适合由数十或数百台服务器组成的数据库,它的路线图中已经包含对 MapReduce 引擎的内置支持。

MongoDB 不适用的场景如下:

1) 要求高度事务性的系统。

2) 传统的商业智能应用。

3) 复杂的跨文档 (表) 级联查询。

## 2.6.1 MongoDB 安装

本节介绍使用 yum 在 CentOS 系统上安装 MongoDB 以及后续配置的方法,但要保证 CentOS 外网连通。

### 1. 配置 yum 源

运行 yum 命令查看 MongoDB 的包信息:

```
yum info mongo*
```

如果提示没有相关匹配的信息,说明 CentOS 系统中的 yum 源不包含 MongoDB 的相关资源,所以要在使用 yum 命令安装 MongoDB 前增加 yum 源,即在 /etc/yum.repos.d/ 目录中增加 \*.repo yum 源配置文件。

文件命名:

```
/etc/yum.repos.d/mongodb-org-3.4.repo
```

命令与内容:

```
vi /etc/yum.repos.d/mongodb-org-3.4.repo
```

```
[mongodb-org-3.4]
```

```
name=MongoDB Repository
```

```
baseurl=https://repo.mongodb.org/yum/amazon/2013.03/mongodb-org/3.4/x86_64/
gpgcheck=1
enabled=1
gpgkey=https://www.mongodb.org/static/pgp/server-3.4.asc
```

配置好 yum 源后，如果配置正确，执行下面的命令便可以查询 MongoDB 相关的信息：

```
yum info mongo*
```

当能看到 MongoDB 3.X 版本字样时，表示 yum 源已经配置正确，可以开始安装了。

## 2. 安装 MongoDB 的服务器端

使用 yum 安装 MongoDB：

```
sudo yum install -y mongodb-org
```

安装完成检查 MongoDB 程序文件：

```
# ls /usr/bin/mongo (tab 键)
mongo          mongoexport    mongooplog     mongos
mongod          mongofiles     mongoperf      mongostat
mongodump      mongoimport    mongorestore   mongotop
```

这些就是 MongoDB 的程序文件。

MongoDB 数据文件默认保存在 /var/lib/mongo 目录，日志文件保存在 /var/log/mongodb，使用 MongoDB 作为运行用户。可以在 /etc/mongod.conf 里修改数据文件和日志文件的路径，参数为 systemLog.path 和 storage.dbPath。

如果修改了运行 MongoDB 的用户，必须确保该用户对 /var/lib/mongo 和 /var/log/mongodb 两个目录有修改权限。

## 3. 运行 MongoDB

可以使用下面的命令运行 MongoDB 进程：

```
sudo service mongod start
```

## 4. 确认 MongoDB 是否启动成功

通过检查 /var/log/mongodb/mongod.log 文件内容检查 MongoDB 是否启动成功，检查下面这行信息：

```
[initandlisten] waiting for connections on port <port>
```

<port> 是在 /etc/mongod.conf 中配置的端口号，默认值是 27017。

也可以使用下面命令让 MongoDB 在系统启动时自动运行：

```
sudo chkconfig mongod on
```

## 5. 停止 MongoDB

使用下面命令停止 MongoDB 进程：

```
sudo service mongod stop
```

## 6. 重启 MongoDB

使用下面命令重启 MongoDB 进程:

```
sudo service mongod restart
```

可以在 /var/log/mongodb/mongod.log 实时查看进程的状态。

## 7. 测试 MongoDB 是否正常运行

进入 mongodb 的 shell 模式:

```
mongo
```

查看数据库列表:

```
show dbs
```

查看当前 db 版本:

```
db.version();
```

## 8. 卸载 MongoDB

### (1) 停止 MongoDB

停止 MongoDB:

```
sudo service mongod stop
```

### (2) 卸载安装包

卸载安装包:

```
sudo yum erase $(rpm -qa | grep mongodb-org)
```

### (3) 删除数据和日志目录

删除数据和日志目录:

```
sudo rm -r /var/log/mongodb
sudo rm -r /var/lib/mongo
```

## 2.6.2 mongod.conf 配置说明

MongoDB 的配置文件为 /etc/mongod.conf, 下面是一个 mongod.conf 示例, 对于每个配置项, 以中文注释项的方式注释在文件内, 具体内容如下。

```
# mongod.conf
```

```
# 日志保存位置
```

```
logpath=/var/log/mongo/mongod.log
```

```
# 以追加方式写入日志
```

```
logappend=true
```

```
# 在后台运行
```

```
fork = true
```



```

# 服务运行端口
#port = 27017

# 数据库文件保存位置
dbpath=/var/lib/mongo

# 启用定期记录 CPU 利用率和 I/O 等待
#cpu = true

# 是否以安全认证方式运行，默认是不认证的非安全方式
#noauth = true
#auth = true

# 详细记录输出
#verbose = true

# 用于开发驱动程序时的检查客户端接收数据的有效性
#objcheck = true

# 启用数据库配额管理，默认每个 db 可以有 8 个文件，可以用 quotaFiles 参数设置
#quota = true
# 设置 oplog 记录等级
# 0=off (默认)
# 1=W
# 2=R
# 3=both
# 7=W+some reads
#oplog = 0

# 动态调试项
#nocursors = true

# 忽略查询提示
#nohints = true
# 禁用 HTTP 界面，默认为 localhost: 28017
#nohttpinterface = true

# 关闭服务器端脚本，这将极大地限制功能
#noscripting = true

# 关闭扫描表，任何查询将会是扫描失败
#notablesan = true

# 关闭数据文件预分配
#noprealloc = true

# 为新数据库指定 .ns 文件的大小，单位为 MB
# nssize = <size>

# 监视服务器的账号 token
#mms-token = <token>

# mongo 监控服务器的名称
#mms-name = <server-name>

```

```
# mongo 监控服务器的 ping 间隔
#mms-interval = <seconds>
```

```
# 复制选项
```

```
# 在复制中, 指定当前是从关系
```

```
#slave = true
```

```
#source = master.example.com
```

```
# Slave only: specify a single database to replicate
```

```
#only = master.example.com
```

```
# or
```

```
#master = true
```

```
#source = slave.example.com
```

以上是默认配置文件中的参数, 更多参数可以使用 `mongod -h` 命令查看:

```
# mongod -h
```

```
Allowed options:
```

```
General options:
```

```
-h [ --help ]
```

显示本使用信息

```
--version
```

显示版本信息

```
-f [ --config ] arg
```

指定启动配置文件路径

```
-v [ --verbose ]
```

显示详细信息

```
--quiet
```

静默输出

```
--port arg
```

端口

```
--bind_ip arg
```

绑定 IP, 可以指定多个

```
--maxConns arg
```

最大并发连接数

```
--logpath arg
```

日志文件路径

```
--logappend
```

日志写入方式

```
--pidfilepath arg
```

pid 文件路径

```
--keyFile arg
```

集群认证私钥, 仅适用于副本集

```
--unixSocketPrefix
```

UNIX 域套接字的可选 arg 路径, (默认是 /tmp)

```
--fork
```

启动服务进程

```
--auth
```

使用认证方式运行

```
--cpu
```

定期显示的 CPU 和 I/O 等待利用率

```
--dbpath arg
```

数据库文件路径

```
--diaglog arg
```

0=off 1=W 2=R 3=both 7=W+some reads oplog 记录等级

```
--directoryperdb
```

每个数据库存储到单独目录

```
--journal
```

记录日志, 建议开启, 在异常宕机时可以恢复一些数据

```
--journalOptions arg
```

日志参数

```
--ipv6
```

启用 IPv6 支持, IPv6 默认关闭

```
--jsonp
```

允许 JSONP 通过 HTTP 访问, 该方式存在安全隐患

```
--noauth
```

不带安全认证的方式

```
--nohttpinterface
```

禁用 HTTP 接口

```
--noprealloc
```

禁用数据文件的预分配, 往往会损害性能

```
--noscripting
```

禁用脚本引擎

```
--notablesan
```

不允许表扫描

```
--nunixsocket
```

禁止 UNIX 套接字监听

```
--nssize arg (=16)
```

为新数据设置 ns 文件的大小

```
--objcheck
```

检查客户端数据的有效性

```
--profile arg
```

0=off, 1=slow, 2=all

```
--quota
```

启用数据库配额管理, 默认每个 db 可以有 8 个文件, 可以用 quotaFiles 参数设置

```
--quotaFiles arg
```

数据库允许的文件数量, 由 --quota 参数使用

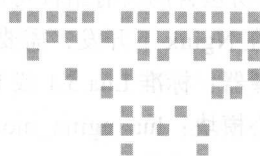
<code>--rest</code>	开启 rest API
<code>--repair</code>	修复所有数据库
<code>--repairpath arg</code>	修复文件的根目录, 默认为 <code>dbpath</code> 指定的目录
<code>--slowms arg (=100)</code>	大于多少秒才算慢查询 (需要与 <code>--profile</code> 配合才会生效)
<code>--smallfiles</code>	最小的文件大小默认值
<code>--syncdelay arg (=60)</code>	与硬盘同步数据的时间, 默认 60 秒, 0 表示不同步到硬盘 (不建议)
<code>--sysinfo</code>	打印一些诊断系统信息
<code>--upgrade</code>	如果必要, 将数据库文件升级到新的格式 (1.0 到 1.1+ 升级时所需的)
<b>Replication options:</b>	
<code>--fastsync</code>	复制选项 从一个 <code>dbpath</code> 快照开始同步
<code>--autoresync</code>	自动同步, 如果从机的数据不是新的则自动同步
<code>--oplogSize arg</code>	<code>oplog</code> 的大小, 单位为 MB
<b>Master/slave options:</b>	
<code>--master</code>	主 / 从配置选项 主模式
<code>--slave</code>	从模式
<code>--source arg</code>	从服务器上指定主服务器地址, 格式为 <code>&lt;server:port&gt;</code>
<code>--only arg</code>	从服务器上指定要复制的数据库
<code>--slavedelay arg</code>	指定从主服务器上同步数据的时间间隔, 单位秒
<b>Replica set options:</b>	
<code>--replSet arg</code>	副本集选项 参数: <code>&lt;名称&gt;[&lt;种子主机列表&gt;]</code>
<b>Sharding options:</b>	
<code>--configsvr</code>	分片设置选项 声明这是一个集群的配置数据库, 默认的端口是 27019, 默认的路径是 <code>/data/configdb</code>
<code>--shardsvr</code>	声明这是集群的一个分片数据库, 默认端口为 27018
<code>--noMoveParanoia</code>	关闭 <code>moveChunk</code> 偏执型数据保存策略

## 2.7 小结

本章向用户介绍了在以 Nginx 为核心的系统中常用的几种数据库, 包括关系型数据库 MySQL、PostgreSQL, 内存型数据库及缓存的 Redis、Memcached, NoSQL 数据库 MongoDB。在一个小型的系统中, 使用 Nginx 和这些数据库及缓存, 可以快速搭建一个系统原型。

对于 Redis 数据库, 需要注意以下两点:

- Redis 本身是单线程的, 因此可以设置每个实例在 6 ~ 8GB 之间, 通过启动更多的实例提高吞吐量。例如, 128GB 的服务器上可以开启 8GB × 10 个实例, 充分利用多核 CPU 的性能。
- 在实际项目中, 为了提高吞吐量, 往往需要使用主从策略, 即数据写到主 Redis, 读取时从从 Redis 上读, 这样可以通过挂载更多的从服务器提高吞吐量, 而且可以通过主从机制, 在叶子节点开启持久化方式防止数据丢失。更大型的系统则需要使用 twemproxy 实现 Redis 集群。



## 第 3 章 Chapter 3

# OpenResty

一个典型的互联网系统或云计算系统大量使用到关系型数据库、非关系型数据库、缓存、内存数据库等，以提供高速、高扩展性的服务，通常组成集群。以前协调这些系统开发是非常麻烦的，现在 OpenResty 提供的 Nginx+Lua+Module 的机制，使我们可以实现快速开发，让开发者着眼于应用，使用同一语言开发。在这种架构的应用领域里，其效率是其他语言和技术不能比拟的。

本章介绍 OpenResty 的组成及安装、配置方法。

### 3.1 OpenResty: 概述

OpenResty 是一个基于 Nginx 与 Lua 的高性能 Web 平台，集成了大量精良的 Lua 库、第三方模块以及大多数的依赖项，用于方便地搭建能够处理超高并发、扩展性极高的动态 Web 应用、Web 服务和动态网关。

OpenResty 通过汇聚各种设计精良的 Nginx 模块（主要由 OpenResty 团队自主开发），从而将 Nginx 有效地变成一个强大的通用 Web 应用平台。这样，Web 开发人员和系统工程师可以使用 Lua 脚本语言调动 Nginx 支持的各种 C 以及 Lua 模块，快速构造出足以胜任 10K 乃至 1000K 以上单机并发连接的高性能 Web 应用系统。

OpenResty 致力于将服务器端应用完全运行于 Nginx 服务器中，充分利用 Nginx 的事件模型进行非阻塞 I/O 通信，不仅仅和 HTTP 客户端间的网络通信是非阻塞的，与 MySQL、PostgreSQL、Memcached 以及 Redis 等众多后端之间的网络通信也是非阻塞的。

因为 OpenResty 软件包的维护者也是其中打包的许多 Nginx 模块的作者，所以 Open-

Resty 可以确保所包含的所有组件可以可靠地协同工作。

使用 Lua 在 Nginx 下开发，需要安装很多支撑库，例如：

- Lua 解释器：标准 Lua 5.1 或 LuaJIT 2.0/2.1，用于对 Lua 语言进行解析。
- Lua 核心模块：lua\_nginx\_module，是 Lua 语言和 Nginx 的桥梁，我们的脚本全部是通过 ngx\_lua 模块和 Nginx 协调起来工作的。其中 Lua 的 VM 也在 ngx\_lua 中工作。
- MySQL 库：异步访问 MySQL 的 Lua 库。
- Redis 库：异步访问 MySQL 的 Lua 库。
- Memcached 库：异步访问 Memcached 的 Lua 库。
- PostgreSQL 库：异步访问 PostgreSQL 的 Lua 库。
- JSON 库：Lua 上的 CJSON 库。
- MySQL RDS 库：MySQL 结果集处理 RDS 库。
- Redis RDS 库：Redis 的结果集处理 RDS 库。
- Drizzle Nginx Module：一个和 MySQL 或 Drizzle 通信的上游服务器。
- .....

这些库都需要分别安装和配置，通过 OpenResty 可以把这些库和 Nginx 打包到一起，让研发者或使用者直接使用，从而省去配置和匹配的麻烦，所以我们推荐使用 OpenResty 进行 Nginx 下 Lua 开发环境的搭建。

## 3.2 OpenResty 的组成

OpenResty 由下面的组件组成。

- 标准 Lua 5.1 解释器；
- Drizzle Nginx 模块；
- Postgres Nginx 模块；
- Iconv Nginx 模块。

所有组件均可以方便地被激活或禁止。绝大部分组件已内置在 OpenResty 安装包中，但也有一部分不包含在内。

上面 4 个模块默认并未启用，需要分别加入 `--with-lua51`、`--with-http_drizzle_module`、`--with-http_postgres_module` 和 `--with-http_iconv_module` 编译选项开启它们。

其余各组件编译选项，可对照 OpenResty 安装说明，按需启用。非必要时，不推荐启用标准 Lua 5.1 解释器，而应尽量使用 LuaJIT 组件。

在 1.5.8.1 版本之前，OpenResty 默认使用标准 Lua 5.1 解释器。所以对于老版本，需要显式地加入 `--with-luajit` 编译选项（1.5.8.1 以后的版本已默认开启）来启用 LuaJIT 组件。

OpenResty 支持的模块如下：

- LuaJIT；

- ArrayVarNginxModule;
- AuthRequestNginxModule;
- CoolkitNginxModule;
- DrizzleNginxModule;
- EchoNginxModule;
- EncryptedSessionNginxModule;
- FormInputNginxModule;
- HeadersMoreNginxModule;
- IconvNginxModule;
- StandardLuaInterpreter;
- MemcNginxModule;
- Nginx;
- NginxDevelKit;
- LuaCjsonLibrary;
- LuaNginxModule;
- LuaRdsParserLibrary;
- LuaRedisParserLibrary;
- LuaRestyCoreLibrary;
- LuaRestyDNSLibrary;
- LuaRestyLockLibrary;
- LuaRestyLrucacheLibrary;
- LuaRestyMemcachedLibrary;
- LuaRestyMySQLLibrary;
- LuaRestyRedisLibrary;
- LuaRestyStringLibrary;
- LuaRestyUploadLibrary;
- LuaRestyUpstreamHealthcheckLibrary;
- LuaRestyWebSocketLibrary;
- LuaRestyLimitTrafficLibrary;
- LuaUpstreamNginxModule;
- PostgresNginxModule;
- RdsCsvNginxModule;
- RdsJsonNginxModule;
- RedisNginxModule;
- Redis2NginxModule;

- RestyCLI;
- OPM;
- SetMiscNginxModule;
- SrcacheNginxModule;
- XssNginxModule。

### 3.3 OpenResty 的安装

本章介绍在 CentOS 6.x 上使用 yum 安装 OpenResty 方法，其他平台的安装方法请到官方网站 (<https://openresty.org/cn/installation.html>) 查看：

对于下列 Linux 发行版的种类和版本号，OpenResty 提供官方的预编译包。

1) RHEL/CentOS。版本号支持的体系结构：

5.x            x86\_64, i386

6.x            x86\_64, i386

7.x            x86\_64

2) Fedora。版本号支持的体系结构：

23            x86\_64, i386

24            x86\_64, i386

25            x86\_64, i386

26            x86\_64, i386

#### 1. 添加资源库

在 CentOS 上使用 yum 安装 OpenResty，需要首先安装资源库，这样就可以方便地安装 OpenResty，以后也可以更新（通过 yum update 命令）。

创建一个名为 /etc/yum.repos.d/OpenResty.repo 的文件，内容如下：

```
[openresty]
name=Official OpenResty Repository
baseurl=https://copr-be.cloud.fedoraproject.org/results/openresty/openresty/
epel-$releasever-$basearch/
skip_if_unavailable=True
gpgcheck=1
gpgkey=https://copr-be.cloud.fedoraproject.org/results/openresty/openresty/
pubkey.gpg
enabled=1
enabled_metadata=1
```

也可以直接运行下面命令添加仓库：

```
sudo yum-config-manager --add-repo https://openresty.org/yum/centos/OpenResty.repo
```



国内用户可以把 baseurl 改成下面的链接，速度会更快：

```
baseurl=https://openresty.org/yum/openresty/openresty/epel-$releasever-
$basearch/
```

或者运行下面命令直接添加仓库：

```
sudo yum-config-manager --add-repo https://openresty.org/yum/cn/centos/OpenResty.
repo
```

## 2. 列出所有包

使用下面命令列出资源库中所有的 OpenResty 包：

```
sudo yum --disablerepo="*" --enablerepo="openresty" list available
```

## 3. 安装

使用下面命令进行安装：

```
sudo yum install openresty
```

使用 yum 安装 OpenResty 可能会因为缺少 GeoIP 库失败，所以需要先运行下面命令安装 GeoIP：

```
yum install GeoIP-devel
```

GeoIP 库的安装可能会因为仓库里没有 Extra 库而失败，所以需要首先添加 Extra 库：

```
yum install epel-release
```

## 4. 测试

运行下面命令启动 Nginx：

```
/usr/local/openresty/nginx/sbin/nginx -p /usr/local/openresty/nginx/
```

在浏览器里输入 <http://127.0.0.1> (或主机 IP)，看到 “Welcome to OpenResty !” 表示已经启动成功。

可以进一步修改 `/usr/local/openresty/nginx/conf/nginx.conf`，测试 Lua 是否正常工作，`nginx.conf` 内容如下：

```
worker_processes 1;
error_log logs/error.log;
events {
    worker_connections 1024;
}
http {
    server {
        listen 8080;
        location / {
            default_type text/html;
            content_by_lua '
                ngx.say("<p>hello, world</p>")
            ';
        }
    }
}
```

```
}
}
```

然后运行下面命令重载配置文件：

```
/usr/local/openresty/nginx/sbin/nginx -p /usr/local/openresty/nginx/ -s reload
```

重载之前可以先测试一下配置文件的正确性：

```
/usr/local/openresty/nginx/sbin/nginx -p /usr/local/openresty/nginx/ -t
```

在浏览器里输入 `http://127.0.0.1:8080`，如果看到了“hello world”就表示可以正常工作了。

也可以使用 CURL 工具测试：

```
curl http://localhost:8080/
```

```
<p>hello, world</p>
```

### 3.4 Nginx 多实例

OpenResty 安装成功后，包里的 Nginx 可以部署多个实例，可以实例化多个不同的服务：或用于对外提供服务，或用于不同的开发任务，或用于学习。

只需要把 OpenResty 中的 Nginx 目录复制一份就可以启动不同的实例：

```
cp -r /usr/local/openresty/nginx /usr/local/openresty/nginx_9090
```

然后修改 `nginx_9090/conf/nginx.conf`，把端口从 8080 修改为 9090，把“hello world”修改为“hello world2”，修改完成后启动实例。

```
/usr/local/openresty/nginx_9090/sbin/nginx -p /usr/local/openresty/nginx_9090/
```

在浏览器里输入 `http://127.0.0.1:9090`，可以得到：

```
hello world2
```

表示新实例启动成功。

### 3.5 小结

本章介绍了 OpenResty 应用，并介绍了 OpenResty 的组成、安装方法；另外，为了方便应用，介绍了在 OpenResty 下多 Nginx 实例的方法。OpenResty 是一个流行的 Nginx 下 Lua 开发解决方案，使用非常广泛。通过本章的介绍，读者可以在后续的学习和工作中掌握 OpenResty 的使用方法，感受其带来的便利性。

## Nginx 核心技术

在开发 Nginx 时, Apache 已经是一个成熟的 Web 服务器了, 性能不错、功能丰富、应用广泛。Nginx 从设计之初就定位为高性能、高可靠性、高扩展性、高并发性的 Web 服务器, 比 Apache 性能更好。Nginx 是基于 HTTP 协议的 Web 服务器, 受 HTTP 协议的制约, 有些问题解决起来比较棘手, 所以采用了一系列技术解决这些问题。

本章介绍这些有特色的技术, 以便我们更了解 Nginx, 知道 Lua 开发的工作机制, 这些技术和思想也可以应用到我们的项目中去。

### 4.1 Nginx 设计目标

Nginx 的设计目标体现 7 个方面, 为了实现这 7 个方面的目标, Nginx 采用了一系列架构和技术, 具体如下。

#### 1. 性能

Nginx 最大的特点是性能优异, 客户接入的并发性出众, 这是 Apache 无法比拟的。这个性能主要体现在网络传输方面。

1) 网络性能: Nginx 服务器整体的吞吐能力。Nginx 在 Linux 上使用了 Epoll 网络模型, 在全异步模式及多进程而非多线程模式的支持下, 可以处理几万至几十万的并发请求。Nginx 在用户干预请求之前, 采用了预处理机制, 保证更多的连接可以接入并被以最快的速度处理。而连接池等技术的使用也进一步提高了接入能力。全异步的处理机制使得响应速度更快, 从而可以处理更多的请求。因为使用进程模式, 减少了大并发情况下线程间切换、休眠等消耗, 使得 CPU 消费很小。而低的内存占用, 也减少了系统内存使用率。

2) 网络效率: 为了提高网络效率, Nginx 使用了长连接 (keepalive) 代替短连接以减少建立、关闭带来的网络交互。同时使用压缩算法提高网络利用率, 减少交互次数。

3) 时延: Nginx 使用了带宽控制技术, 使各会话之间带宽尽量相等, 保证每个连接都尽量通信, 同时保证相同请求链路间低带宽和高带宽变化不会太大。全异步的模式也保证了每个请求都会以最快的速度被处理而不会被阻塞, 而且每个请求的处理时间是可以预期的, 保证了每个连接时延都是均等且可预期的, 总体时延是相对较低的。

## 2. 可靠性

可靠性是 Nginx 第二大特点。Nginx 服务本身采用了主从机制以看门狗形式管理工作进程, 一旦有工作进程崩溃, 会马上启动新的进程代替。Nginx 支持负载均衡机制, 可以平行增加冗余点。

## 3. 伸缩性

Nginx 使用了组件技术, 可以减少或增加使用的组件, 或使用自行开发的新组件, 介入到 HTTP 请求处理的中间环节, 改变处理行为。Nginx 提供了 6 个核心模块组成整个服务, 核心模块中还可以串行增加新的模块以改变服务。

## 4. 简单性

多组件以及多阶段的方式使一个 HTTP 处理过程被分为了 11 个小的阶段, 每个阶段都可以非常简单, 这使得每个阶段都容易理解和实现, 也容易验证。Nginx 的这种组件 + 分阶段的模式比通常系统中的模块化更进了一步, 更加细化。直接的好处就是使 HTTP 处理过程变成了流水线模式, 每个模块只是流水线上的一道环节的加工者, 工作相对非常简单。一个间接的好处是, 因为分了阶段, 就要求模块间要按接口开发, 接口相对固化、简化、统一, 模块更高效、稳定。

## 5. 可修改性

可修改性指的是当前架构下对系统功能修改的难易程度。对于 Nginx 这种定位专用的 Web 服务器, 还需要具备动态修改配置、动态升级、动态部署的能力。动态修改配置指的是根据业务情况实时修改配置而不需要重启服务。动态升级指的是不重启服务而将 Web 服务器升级到新版本的能力。可修改性还可以理解为可扩展性、可定制性、可重用性等。

## 6. 可见性

可见性指的是在 Web 服务器的应用场景中, 系统的关键组件的运行情况可以被监控, 如网络吞吐量、网络连接数、缓存使用情况等。通过这些数据的监控可以及时修改系统配置和服务配置, 以改善服务性能。

## 7. 可移植性

可移植性是指跨平台能力, 通常在大型的 Web 应用中, 操作系统是 Linux 和 UNIX, Nginx 使用 Epoll, 可以发挥最大性能。同时 Nginx 也支持 Windows, 但是在 Windows 上使

用的是 `select`，性能比较低，适合应用于小型应用，如专业型行业平台的 Web 容器，而不是大型 Web 应用。

## 4.2 Nginx 架构

Nginx 使用了很多提升稳定性和性能的架构，这些技术都非常有效，虽然有的技术本身看起来比较简单。

总体来看，Nginx 使用事件驱动的服务模型。为此，Nginx 在它的模块机制中专门定义了 `event` 模块实现事件驱动。在事件的基础上，Nginx 使用了多阶段的异步模型。Nginx 的异步模式将处理过程划分为阶段，进一步将异步的作用范围缩小。把一个处理过程的过程（如 HTTP 请求）划分为 7 个、9 个或 11 个阶段，每一个阶段都异步处理，将请求和处理结果异步化处理。将请求多阶段处理，可以进一步控制每个请求的总体处理时间，因为每个阶段都细化，不会出现某个阶段过多占用 CPU 处理时间的问题。

这种多阶段的机制是保证 Nginx 能大并发处理多请求的基础，同时为使用模块介入请求处理提供了基础，Lua 开发就是在这几个阶段中实现的。

管理进程和工作进程的机制使 Nginx 可以充分利用多处理器机制，充分利用了 SMP 机制的硬件资源，又可以减少过多线程、进程的调度开销。管理进程作为工作进程的管理者，监控工作进程的工作状态，可以做到动态升级，当工程进程终止时，管理进程可以快速启动新的进程，保证系统的可用性。

对于高并发下的多工程进程容易引起的“惊群”以及负载均衡问题，Nginx 都做了比较好的处理。同时对于其他容易引起更多系统资源消费的访问，例如，每个请求阶段中频繁出现的时间比较操作，Nginx 做了时间缓存机制。在其他通用的内存管理、连接池、跨平台管理等方面，Nginx 都做得比较好。下面分别介绍这些架构和技术。

### 4.2.1 事件驱动

Nginx 是事件驱动型服务，注册各种事件处理器以处理事件。对于 Nginx，事件主要来源于网络和磁盘。事件（`event`）模块负责事件收集、管理和分发事件，其他的模块都是事件的处理器和消费者，会根据注册的事件得到事件的分发。

因为 Nginx 是 Web 服务器，而 HTTP 协议是基于 TCP 的，所以，Nginx 的网络事件基本上来自于 TCP 网络。因为 Nginx 要跨平台，支持多个操作系统，所以，需要在不同的操作系统上支持不同的网络模型和事件机制。目前支持的事件机制如下：

- Linux 内核 2.6 以前版本和大部分 UNIX 操作系统使用 `poll` 机制（`ngx_poll_module` 事件模块）或 `select` 机制（`ngx_select_module` 事件模块）。
- Linux 内核 2.6 以上版本使用 `epoll` 机制（`ngx_epoll_module` 事件模块）。`epoll` 是 Linux 操作系统上最强大的事件管理机制。

- Windows 操作系统上使用 select 机制。
- Solaris 10 使用 eventport 机制 (ngx\_eventport\_module 事件模块)。

在不同的操作系统上, 在 nginx.conf 的 event{} 块中配置相应的事件模块, 就可以启用对应事件模型, 而且可以根据应用场景随时切换事件模块, 这也实现了 Nginx 的跨平台机制。这个具体的 event 模块被核心的 ngx\_events\_module 管理, ngx\_events\_module 是核心模块。

Nginx 为不同的操作系统和不同内核版本提供了共 9 个事件模块, 分别为 ngx\_select\_module、ngx\_eventport\_module、ngx\_epoll\_module、ngx\_poll\_module、ngx\_devpoll\_module、ngx\_kqueue\_module、ngx\_aio\_module、ngx\_rtsig\_module, 以及一个 Windows 版本 ngx\_select\_module。

图 4-1 描述 Nginx 事件处理的模型, 即事件、事件管理器和事件消费者之间的一个概貌。

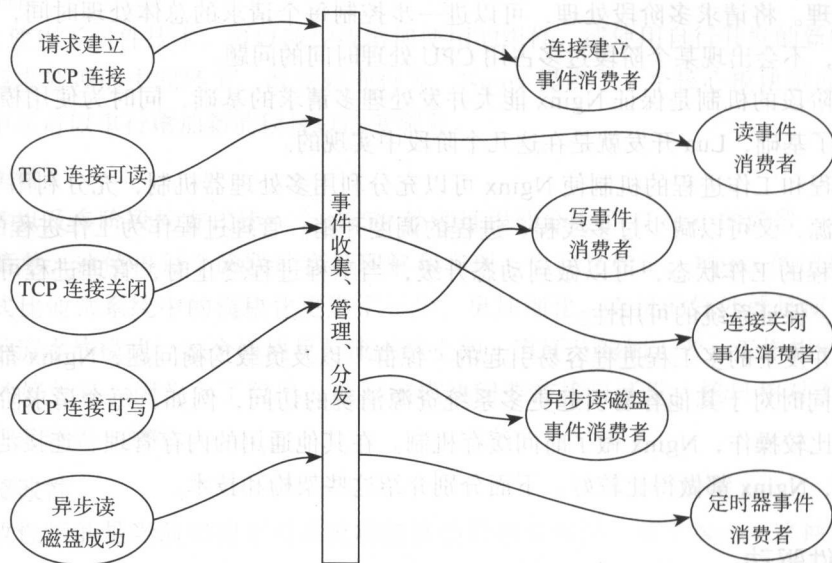


图 4-1 Nginx 事件处理模型

图 4-1 中所示的模型分为事件、事件管理器、事件消费者。事件是由生产者产生的, 事件管理器负责收集、管理、分发, 事件消费者使用和消费这些事件。而事件, 不单单是网络事件, 还有 Nginx 自定义的事件。

Nginx 定义了丰富的事件, 这些事件的消费者全部是模块, 模块可以注册不同的事件, 而事件可以细化到一个操作的不同阶段。图 4-1 中列出了 5 个事件, 被 5 个事件消费者处理。这种细化的事件加上非阻塞的处理, 可以使响应速度大大加快。区别于传统的 Web 处理机制, 每个事件消费者都不去阻塞事件, 处理完自己的事件后如果条件不满足, 可以进入休眠状态, 从而简化了开发过程的数据同步、进程同步、重入等问题。如果要管理一个



事件,就需要首先注册一个 handler,这个 handler 负责这个事件处理,ngx\_lua 正是利用注册 handler 实现的自定义开发,对 HTTP 请求处理过程进行干预。

Nginx 的事件机制是完全的事件驱动,与传统 Web 服务不同。传统服务每个事件消费者都独占一个进程资源,而 Nginx 的完全事件驱动只是被事件分发者进程短期调用。每个用户请求产生的事件都会得到及时的响应,每个消费者不允许阻塞程序,不允许消费者使进程变为休眠状态或等待状态。

### 4.2.2 异步多阶段处理

Nginx 的异步多阶段处理是基于事件驱动架构的。Nginx 把一个请求划分成多个阶段,每个阶段都可以由事件、分发器来分发,注册的阶段管理器(消费者、handler)进行对应阶段的处理。所以,Nginx 的阶段划分相当于人为创造了很多事件。例如,获取一个静态文件的 HTTP 请求可以划分为下面的几个阶段。

- 1) 建立 TCP 连接阶段:收到 TCP 的 SYN 包。
- 2) 开始接收请求:接收到 TCP 中的 ACK 包表示连接建立成功。
- 3) 接收到用户请求并分析请求是否完整:接收到用户的数据包。
- 4) 接收到完整用户请求后开始处理:接收到用户的数据包。
- 5) 由静态文件读取部分内容:接收到用户数据包或接收到 TCP 中的 ACK 包,TCP 窗口向前划动。这个阶段可多次触发,直到把文件完全读完。不一次性读完是为了避免长时间阻塞事件分发器。
- 6) 发送完成后:收到最后一个包的 ACK。对于非 keepalive 请求,发送完成后主动关闭连接。
- 7) 用户主动关闭连接:收到 TCP 中的 FIN 报文。

只有多阶段划分,才能更好地实现异步处理。当一个事件被分发到事件消费者中处理时,消费者只是处理完了一个阶段的事件,只有一个请求的所有阶段都被处理完成,一个完整的流程才算走完,一个完整的请求被划分成很多个过程,有的过程(如读取)还会进一步被多次调用。每一个事件、每一个阶段均由 event 模块负责调用和激活。event 模块监听系统内核消息,以激活 Nginx 事件。

这种异步多阶段处理模式的好处是高效。这将极大地提高网络性能,使每个进程都全力运转,不会或者尽量少地出现进程休眠情况。一旦有进程休眠,必然减少并发处理事件的数量,会降低网络性能。如果网络性能无法满足业务需求,将会增加进程,进程数目过多会造成进程间更多地切换,这也会消耗系统 CPU 资源,反过来影响网络性能。

Nginx 中的阶段划分方法使用了若干种有效的方法,从上往下依次如下。

- (1) 将系统本身的事件和网络异步事件划分为阶段
- 如上面的 7 个阶段划分,这是网络模型通常的几个阶段,而现有的网络模型都在这个基础上划分为若干个事件。



## (2) 将阻塞进程的方法按照相关的触发事件分解为两个阶段

分析可能导致进程休眠的方法或系统调用，将阻塞的方法改成非阻塞的方法，调用这个方法的过程定义为第一个阶段。增加第二个阶段，用于处理非阻塞方法回调。非阻塞方法的返回结果是第二个阶段的触发事件。例如，将阻塞 `send` 接口改成非阻塞 `send` 接口。以 `send` 为节点将 `send` 和 `send` 之后划分为两个阶段。

## (3) 将阻塞方法按调用时间分解为多个阶段调用

在阻塞方法不能按上面方法划分多阶段情况下（如触发事件不可以被捕获），使用按照执行时间拆分方法。

Nginx 主要处理网络收发任务，`epoll` 的网络处理能力非常强大，支持完整的异步模式。一般无法划分的阻塞方法是发生在文件 I/O 上。在不支持异步文件 I/O 的系统上，如果读取 10MB 数据将会遇到文件不连续、中间不定时会等待的情况。这时将文件读取划分为小块读取，如每次读取 10KB，这样每一块读取的时间都比较均匀，这个事件接收器不会占用进程多少时间，系统有机会处理其他的事件。但是没有读取完成的事件，通过将读取和网络发送直接关联，就可以在发送完成事件里知道上次读取已经完成并且已经发送出去，需要读取下一个 10KB 了。在不使用网络的情况下，可以将读取和一个专用定时器关联起来，因为每 10KB 的读取时间相对可控，比较均匀，所以时钟的命中率就比较高，一旦时钟检测到数据读取完成，就可以触发对应第二阶段的事件。

## (4) 在必须等待的情况下，使用定时器切分阶段

例如，我们经常在代码中做状态和标志检测的工作，直到某个条件满足时才往下执行，这种情况使用定时器检测标志，如果标志位不满足就立刻归还控制权，同时继续加入下一个定时器事件。

## (5) 阻塞方法无法继续划分，则必须使用独立的进程执行这个阻塞方法

如果某个阻塞方法没有提供非阻塞接口，则将其划分为阻塞方法调用前和阻塞方法调用后，而调用前需要使用独立进程调度。

Nginx 提供的 API 以及 `ngx_lua` 提供的 API 和库都是非阻塞的，也都使用多阶段的异步机制。我们只是在此基础上开发应用，一般不会遇到阻塞的情况。但如果我们使用 Lua 原生库，则可能会碰到这个情况，典型的就文件类 API，这时我们也需要使用上面的方法进行多阶段划分。再如，我们的线程函数里就可能出现第 4 种情况，需要用对应的方法处理，可使用时钟解决这个问题。对于我们使用非 `OpenResty` 的库，使用其他的库，而库函数没有非阻塞接口，则会遇到第 5 种情况的问题，我们需要使用对应方法解决。否则，我们的代码将阻塞 Nginx 的工作进程，降低系统整体性能。这与 Nginx 和我们的初衷是违背的，需要注意。

Nginx 中的定时器是由 Nginx 实现的，非内核态的，而且使用内存缓冲时间实现高性能读取，所以使用定时器时的性能是可以保证的。

### 4.2.3 模块化设计

模块化设计是 Nginx 中重要的架构，除了少量的核心代码，其他功能都是在模块中实现的，新功能的扩展是通过按照标准接口和数据结构开发新模块实现的，核心代码和核心模块不需要改动。模块化设计的好处是给 Nginx 带来了更好的扩展性、可靠性。

#### 1. Nginx 模块化设计的特点和技术

##### (1) 使用模块接口

所有的模块遵循统一的接口设计规范，接口设计规范定义在 `ngx_module_t` 中，这样使接口简化、统一、可扩展。

##### (2) 配置功能接口化

Nginx 将配置信息也定义和开发成模块，配置模块专注于配置的解析和数据保存，是唯一一个只有一个模块的模块类型。`ngx_module_t` 接口中定义了一个 `type` 成员，用于描述和定义模块类型。配置模块是 `ngx_conf_module`，是其他模块的基础模块，因为 Nginx 模块全部使用配置模块来定义和配置，依赖于配置模块。

##### (3) 模块分层设计

Nginx 设计了 6 个基础类型模块（称为核心模块），实现了 Nginx 的 6 个主要部分，以及 HTTP 协议主流程。这样设计的目的是使框架程序只关注于如何调用核心模块，核心模块实现 Nginx 的核心功能，实现了第一层的流水线。核心模块之外是非核心模块，由对应的核心模块进行初始化和调用。这些模块可以动态添加，通过重新编译包含进 Nginx，通过配置文件将模块使能，给 Nginx 带来了扩展性、灵活性。

event 模块、HTTP 模块、mail 模块的非核心模块中有一个对应的 core 模块，代理核心模块的行为，例如，event 模块均由 `ngx_events_module` 模块定义，加载由 `ngx_event_core_module` 模块负责。这 3 种模块不直接执行框架代码的操作，而是由对应的 core 代理模块实现，它们只重新定义了下面可以管理的模块。

##### (4) 核心模块接口简单化

核心模块的接口非常简单，将 `ngx_module_t` 中的 `ctx` 上下文实例化为 `ngx_core_module_t` 结构，该结构是以配置项的解析为基础的。`nginx.conf` 中解析出来的配置项会放到这个数据结构，通过提供的 `init_conf` 回调使用解析出的配置项初始化核心模块。Nginx 框架允许核心模块接口、功能可自定义，这样就可以使核心模块自己定义新的模块类型，所以就有了 event、mail 和 HTTP 模块的模块簇。这使得 Nginx 可以灵活扩展、不停止服务升级、动态配置。

#### 2. 核心模块

核心模块是 `ngx_core_module`。目前核心模块中有 6 个模块：`ngx_core_module`、`ngx_events_module`、`ngx_openssl_module`、`ngx_http_module`、`ngx_mail_module`、`ngx_errlog_module`。核心的功能都封装在这些核心模块中，框架代码只是这些模块的使用者，可以专

注于主处理流程开发。模块使系统模块化，可以分别升级和进化，使系统功能和稳定性提升，但又不影响整体功能和稳定性。这是一种典型的模块化面向接口编程技术。

这 6 个核心模块分别代理 6 类模块，它们是其他子模块或非核心模块的管理者：

- ngx\_events\_module：管理所有事件类模块。
- ngx\_http\_module：管理所有 HTTP 类型模块。
- ngx\_mail\_module：管理所有邮件类型模块。
- ngx\_errlog\_modul：管理所有日志类模块。
- ngx\_openssl\_module：管理所有 TLS/SSL 模块。
- ngx\_core\_module：管理配置等全局模块。

模块化使 Nginx 可以实现动态可配置性、动态扩展性、动态可定制性，可以实现不停止服务、不重启服务实现更新和添加。

这 6 个核心模块只是定义了 6 类业务的业务流程，具体的工具并不由这些模块执行，例如，event 模块由 ngx\_events\_module 定义，但是由 ngx\_event\_core\_module 模块加载；HTTP 模块由 ngx\_http\_module 定义以及加载，但业务核心逻辑以及具体请求应该调用哪个 HTTP 模块处理由 ngx\_http\_core\_module 实现。

图 4-2 描述了常用模块之间的关系。

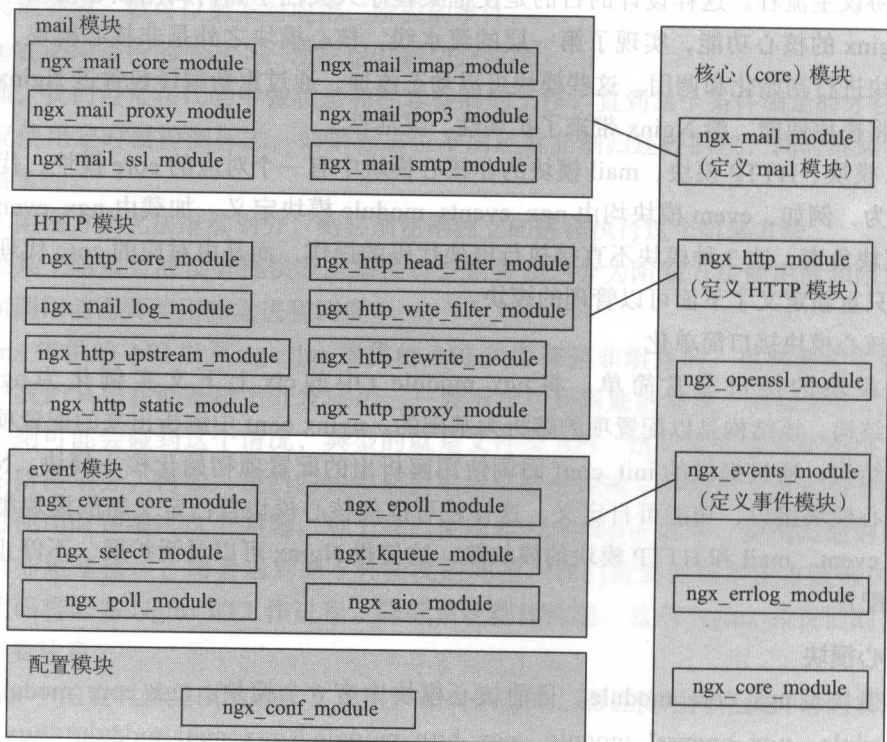


图 4-2 Nginx 常用模块之间的关系

配置模块和核心模块都是框架所面对的一层重要模块。其他模块则像部门经理，各自统领一个部门，负责不同的业务实现。

#### 4.2.4 管理进程、工作进程设计

为支持现在流行的多 CPU 和多核架构，Nginx 使用了管理进程 + 工作进程的设计。管理进程作为工作进程的管理进程和父进程，还可以带来高可靠性：工作进程终止，管理进程可以及时启动新的实例接替。这种模式的优点体现在以下 3 个方面。

1) 充分利用多核系统的并发处理能力。Nginx 中所有的工作进程都是平等的，并且可以在 `nginx.conf` 中将工作进程和处理器一一绑定，这样配合负载均衡机制，不会让某核繁忙，整体处理能力得到提高。

现代的服务器都支持多处理器架构，且处理器内还是多核心架构，只有多进程和多线程机制才能发挥硬件体系的最佳性能。而设计合理的多进程模式比多线程模式性能要好些，因为滥用线程带来的线程切换开销也是不容忽视的。

2) 负载均衡。工作进程间通过进程间通信实现负载均衡，请求容易被分配到负载较轻的工作进程中，这将提高系统的整体性能。

3) 方便状态监管。管理进程任务很轻，只负责启动、停止、监控工作进程，当某个工作进程不正常工作时，可以立即启动新的进程接管。同时管理进程支持服务运行中的程序升级、配置项修改等操作。使系统整体具备了动态扩展性、动态可定制性、动态可进化性。

Nginx 管理 / 工作多进程模式如图 4-3 所示。

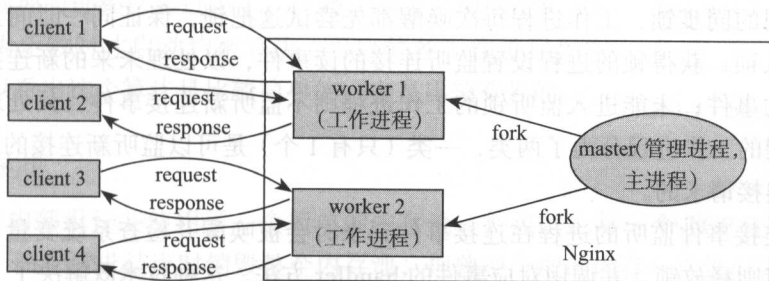


图 4-3 Nginx 管理 / 工作进程模式

管理进程向用户提供了命令行服务，包括启动、停止、重载配置文件、平滑升级程序等。管理进程运行时需要较大的权限。一般会使用 `root` 用户启动管理进程，而工作进程权限较小，使管理进程可以完全管理工作进程。当任意一个工作进程出现错误 `coredump` 时，管理进程会立刻启动新的工作进程继续服务。

多工作进程处理请求可以提高服务的健壮性，即若有工作进程意外退出，则会有新的工作进程启动代替，更可以充分利用 SMP 多核架构，实现真正的多核并发处理。Apache 的每个进程在一个时刻只处理一个请求，因此在多任务处理阶段，Apache 的进程数或线程数

要设置得很多，一般一台服务器可以达到几百个进程，这种情况下大量的进程间切换将带来无谓的系统资源消耗。Nginx 的工作进程之间处理并发请求时几乎没有同步锁的问题，而工作进程采用全异步的操作模式，处理速度快，所占内存非常小。所以，当 Nginx 的进程数与 CPU 核心数相等时，进程间切换的开销是最小的。

### 1. “惊群”问题

管理进程 + 工作进程模式有很多优点，同时也有一些问题需要解决。

Nginx 里的工作进程一般是按系统 CPU 核数配置的，有多少个 CPU 核心，就会配置多少个工作进程，工作进程启动时就会利用 fork 函数创建多少个工作进程，并且所有的工作进程都监听在 nginx.conf 内配置的监听端口上，这样可以充分利用多核机器的性能。网络事件通过底层的 events 模块管理，当客户端连接请求到来时，一个新连接事件会上报，各个工作进程就会发生对事件的抢夺，这就是“惊群”问题。工作进程越多，问题越明显，这会造成系统性能下降，所以，必须避免“惊群”问题。详细来说，“惊群”问题的典型场景是这样的：在没有用户请求的时候，所有的工作进程都在休眠，此时，一个用户向服务器发起了连接请求，例如，在 epoll 模式下，内核在收到了 TCP 的 SYN 包时，会激活所有休眠的工作进程，最先接收连接请求的工作进程可以成功建立新连接，其他工作进程的接收会失败。这些失败的唤醒是不必要的，引发了不必要的进程上下文切换，增加了系统开销，这就是“惊群”问题。

Nginx 应用层制定了一个机制解决这个问题：规定同一时刻只能有唯一一个工作进程监听 Web 端口，这样，新的连接事件只能唤醒唯一一个工作进程。内部的实现实际上是使用了一个进程间的同步锁，工作进程每次唤醒都先尝试这把锁，保证同一时间只有一个工作进程可以进入锁，获得锁的进程设置监听连接的读事件，以处理未来的新连接请求，并处理已连接上的事件；未能进入监听锁的工作进程则不监听新连接事件，只处理已连接上的事件，将唤醒的工作进程分为了两类，一类（只有 1 个）是可以监听新连接的，另一类是正常处理已有连接请求的。

设置了连接事件监听的进程在连接事件到来时会被唤醒并检查系统变量，发现新连接队列中有连接则释放锁，并调用对应事件的 handler 方法。这种技术既解决了“惊群”问题，也避免了一个进程过长占用锁使新连接得不到及时处理的问题，接收了一个连接后，把连接放入队列后马上释放锁，如果恰巧有新连接马上进来，则会由一个新的工作进程接收连接，起到一定的负载均衡作用。放入队列的请求事件会在后续阶段处理。

### 2. 负载均衡

Nginx 的负载均衡可以从两个层面来讲。

#### (1) 系统级的负载均衡

如电子商城一类的大型网站，需要多台 Web 服务器组成集群以应对海量的访问，需要在不同的 Web 服务器之间实现负载均衡，一般有如下的做法。



1) 系统级的负载均衡, 实现方法是使用一个 Nginx 服务通过 upstream 机制将请求分配到上游后端服务器, 而这里可以使用模块内置的一些负载均衡机制将请求均衡地分配到服务器组中。

2) 使用一个单独的 Nginx 服务以自定义负载均衡算法实现代理模式, 实现负载均衡集群。

### (2) 单 Nginx 服务内部工作进程间的负载均衡

不让某个进程“累死”, 其他的进程“闲死”, 才能发挥系统的最佳性能。

这里讲的是管理 / 工作进程模式下工作进程内的负载均衡机制。

Nginx 内部有一个 `ngx_accept_disabled` 变量, 设置的是负载均衡的阈值, 是一个整数数值。`events{}` 中的 `worker_connections` 参数, 用于设置每个工作进程的连接数, 这个连接数会影响连接池的大小, 连接池在内部是一个数据结构, 内部分为 `free_connections` 和 `connections`, 用完的连接都插入 `free_connections` 头部, 新的连接从 `free_connections` 尾部取出一个连接, 然后放入 `connections`, 这些数据都是链表操作。

图 4-4 描述了 `ngx_connection` 的结构。

`ngx_accept_disabled` 是一个进程内的全局变量, 在 Nginx 启动时是负数, 每次接收一个新连接时都会赋值, 值为连接总数的 7/8。当本变量值为负数时, 不会进行负载均衡操作, 会参与到新连接的接收尝试中, 尝试获取同步锁; 当值为正数时, 表示连接已经过多, 则会放弃一次争夺, 并将值减 1。值为正数表示本进程处理的进程已经过多了, 已经达到了上限的 7/8, 所以, 只有值为正数时才启动均衡算法, 可以使各工作进程相应地均衡。可以看出这个算法是比较简单的, 也是比较有效的。

files
file n
connections
connection n
free connections
free connections n
reusable_connections_queue

图 4-4 `ngx_connection` 结构

## 4.2.5 内存池

Nginx 在内部设计并使用了一个简单的内存池, 特点是, 每一个 TCP 连接建立时分配一个内存池, 而在请求结束时销毁整个内存池, 把曾经分配的内存一次性归还给操作系统。而它不负责回收内存池中已经分配出去的内存, 这些内存由请求方负责回收。内存池减少了分配内存带来的资源消耗, 同时减少了内存碎片。一般 Nginx 内部的内存池有以下两种模式: ①申请了永远保存; ②申请了, 请求结束全部释放。对于内存的使用, 有缓冲写满了从头覆盖使用的方法等。

这种内存池是内部使用的, 对于基于 C 的模块可以使用这个内存池机制, 内部提供了 `ngx_palloc`、`ngx_pnalloc`、`ngx_pcall` 这 3 个系统调用模块用于内存申请。内存池机制是为没有垃圾回收机制的 C 语言提供的一个补充机制, 因为在 C 语言中容易出现内存泄漏问题。当内存申请和释放逻辑比较远时, 容易出错, 例如释放两次这种异常。内存池在开发上可以降低使用错误的机率, 模块开发者只需要关注内存的使用情况, 释放则由内存池来负责。

对于 Lua 开发, 这种机制并不能直接使用。Nginx 的这种机制提高了系统整体的可用性, 方便了 Nginx 模块的开发, 我们在 Lua 开发中用到的很多模块就使用了内存池机制。

#### 4.2.6 连接池

Nginx 为了减少反复创建 TCP 连接以及创建套接字的次数, 从而提高网络响应速度, 在内部提供了连接池机制, 在众多配置命令里也可以经常看到连接池的配置选项和单独命令。

连接池在 Nginx 启动阶段, 由管理进程在配置文件中解析出来对应的配置项, 配置项放到配置结构体中。在 event 核心模块 `ngx_events_module` 初始化事件模型时, `ngx_event_core_module` 模块第一个被初始化, 这个模块将根据配置结构体中的连接池大小配置创建连接池, 如果没有配置项, 则使用系统默认值创建连接池。注意, 配置指令中 `worker_connections` 配置的连接池大小是工作进程级别的, 所以实际的连接池大小是 `worker_connections*worker_processes`。

所有使用连接池的接口都有 `keepalive()` 方法, 会将一个连接放入连接池中, 对于应用来讲, 连接将得到 `close` 状态, 而连接实际没有释放。

Nginx 内部封装了两个连接池方法: `ngx_get_connection` 和 `ngx_free_connection`, 用于模块开发者使用内置的连接池, 所以很多 Nginx 配套的模块, 都使用了这个连接池。在后面的实战章节里, 我们将会看到各种连接缓存、数据库的库都支持使用连接池, 特别是 OpenResty 系列数据访问组件, 支持内置的连接池。

#### 4.2.7 时间缓存

Nginx 内部提供了很多时间函数, 而且内部的操作大多数要控制超时值, 需要使用当前时间进行判断。由于对 OS 的时间函数 `gettimeofday` 的调用是内核态的系统调用, 如果频繁调用会降低系统可用性, Nginx 在自己内部对系统时间进行了缓冲, 避免频繁进行系统调用, 内部访问时间实际上访问了内存中的几个变量。

`nginx.conf` 中的 `timer_resolution` 配置可以指定多长时间更新一次时间缓存。这也是一个降低系统资源占用的细节。因为, 考虑到大并发大任务下的处理过程中, 要处理很多的过程, 那么这个占用带来的开销是不容忽视的, 而其他的系统往往又会忽视这个因素。

时间缓存在系统初始化时被赋值, 另一个修改的机会是在 `ngx_epoll_process_events` 调用 `epoll_wait` 返回时有可能更新。

因为 Nginx 的多工作进程机制, 可能导致时间缓存读写不一致问题, 即前一个进程在读时间缓存时正好被中断了, 而时间缓存又被另一个进程因为 `ngx_epoll_process_events` 导致了时间更新了, 导致前后读取不一致。所以采用 64 个缓存时间, 引入时间缓存数组 (共 64 个成员), 每次都更新数组中的下一个元素; 读取时间缓存时, 也是读取最新的时间, 从而实现读写一致性。这是内部实现的机制, 对于上层的应用开发是透明的。



### 4.2.8 延迟关闭

延迟关闭,即当 Nginx 要关闭连接时,并不马上关闭连接,而是先关闭 TCP 连接的写操作,等待一段时间后再关掉连接的读操作。假设有这样一个场景: Nginx 在接收客户端请求时,可能由于客户端或服务端出错,要立即响应错误信息给客户端, Nginx 在响应错误信息后,大部分情况下需要关闭当前连接。 Nginx 执行完 `write()` 系统调用把错误信息发送给客户端, `write()` 系统调用返回成功并不表示数据已经发送到客户端,有可能还在 TCP 连接的写缓冲区里。如果紧接着执行 `close()` 系统调用关闭 TCP 连接,内核会首先检查 TCP 的读缓冲区里有没有客户端发送过来的数据留在内核态没有被用户态进程读取。如果有则发送给客户端 RST 报文关闭 TCP 连接,丢弃写缓冲区里的数据;如果没有则等待写缓冲区里的数据发送完毕,然后经过正常的 4 次分手报文断开连接。所以,若在某些场景下 TCP 写缓冲区里的数据在 `write()` 系统调用之后到 `close()` 系统调用执行之前没有发送完毕,且 TCP 读缓冲区里面还有数据没有读,则 `close()` 系统调用会导致客户端收到 RST 报文且不会拿到服务端发送过来的错误信息数据。客户端就会经常在没有错误信息的情况下被重置连接。

在这个场景中,关键点是服务端给客户端发送了 RST 包,导致自己发送的数据在客户端被忽略掉了。所以,解决问题的重点是,不让服务端发 RST 包。服务端发送 RST 包是因为关掉了连接,关掉连接是因为不想再处理此连接了,也不会有任何数据产生。对于全双工的 TCP 连接来说,只需要关掉写连接就行了,读可以继续进行,只需要丢掉读到的任何数据就可以了,当关掉连接后,当客户端再发过来数据时,就不会收到 RST。设置一个超时时间,在这个时间过后,就关掉读连接,客户端再发送的数据就会被忽略掉,服务端会在超时时间内关掉读端。通过 `lingering_timeout` 选项设置这个超时值,如果在 `lingering_timeout` 时间内还没有收到数据,则直接关掉连接。 Nginx 还支持设置一个总的读取时间,通过 `lingering_time` 设置,这个时间也就是 Nginx 关闭写之后,保留套接字的时间,客户端需要在这个时间内发送完所有的数据,否则 Nginx 在这个时间过后,会直接关掉连接。 Nginx 支持配置是否打开延迟关闭选项,通过 `lingering_close` 选项配置。延迟关闭的主要作用是保持更好的客户端兼容性,但是需要消耗更多的额外资源,因为连接会一直占用。

### 4.2.9 跨平台

Nginx 使用 C 语言开发,开发过程中可减少平台相关调用。在关键的网络部分,因为使用了模块技术, Nginx 按不同平台提供了不同的模块,可以在 `nginx.conf` 中配置使用,适应了不同平台的环境;同时在内部重新封装了各种数据结构和容器代码,重新封装了日志。所以, Nginx 可以在各种平台上运行,实现跨平台工作。

#### 4.2.10 HTTP 模块管道过滤模式

Nginx 中定义了一种 HTTP 过滤模块。过滤模块有输入端和输出端,输入端和输出端有统一的接口。过滤模块按照配置时的次序依次连接,组成一个过滤链,每个模块处理接收

到的数据，处理完成后输出到下一个模块，每一个模块都增量式地处理数据，可以正确处理完整数据流的一部分。

这种模式允许把整个 HTTP 过滤系统输入/输出简化为可以组合的机制。用户可以将任意的过滤模块按照业务要求组合起来，实现特定的要求。开发一个新的模块后，可以简单地将其添加到现有过滤系统中，提高了可验证性和可测试性；可以灵活地变动这个过滤模块流水线以验证功能，验证完毕可以使系统方便地扩展 HTTP 过滤模块，提高了扩展性，更便于开发和验证。

#### 4.2.11 keepalive

keepalive 是 HTTP 长连接，为了提高传输效率，HTTP 协议中定义了这个特性。keepalive 可以有效提高网络效率，所以 Nginx 对这个协议特性进行了运行，在配置指令以及 API 中，可以大量看到相关的配置和 API。

在 Nginx 中，HTTP 1.0 与 HTTP 1.1 也支持长连接。HTTP 请求是基于 TCP 传输层协议的，客户端发起请求前，需要与服务端建立 TCP 连接，每一次的 TCP 连接需要 3 次握手确定，如果客户端与服务端之间的网络环境比较差，则这 3 次交互占用的时间会比较多，而且 3 次交互也会消耗网络流量。连接断开时，需要进行 4 次交互。HTTP 协议是请求应答式的，如果知道每个请求头与响应体的长度，那么可以在一个连接上面执行多个请求，这就是长连接，前提条件是先确定请求头与响应体的长度。对于请求来说，如果当前请求包含 body，如 POST 请求，那么 Nginx 需要客户端在请求头中指定 Content-Length 表明 body 的大小，否则返回 400 错误，也就是说，请求体的长度是确定的。HTTP 协议中关于应答包 body 的长度定义如下。

- 对于 HTTP 1.0 协议来说，如果响应头中有 Content-Length 头，则根据 Content-Length 的长度可以知道 body 的长度，客户端在接收 body 时，可以依照这个长度接收数据，接收完后，表示这个请求完成了。而如果响应头中没有 Content-Length 头，则客户端会一直接收数据，直到服务端主动断开连接，才表示 body 接收完了。
- 对于 HTTP 1.1 协议来说，如果响应头中的 Transfer-encoding 为 chunked 传输，则表示 body 是流式输出，body 会被分成多个块，每块的开始会标识出当前块长度，此时，body 不需要通过长度指定。如果响应头中的 Transfer-encoding 为非 chunked 传输，而且有 Content-Length，则按照 Content-Length 接收数据；否则，如果响应头中的 Transfer-encoding 为非 chunked 传输，并且没有 Content-Length，则客户端接收数据，直到服务端主动断开连接。

可以看到，除了 HTTP 1.0 不带 Content-Length 以及 HTTP 1.1 非 chunked 传输不带 Content-Length 外，body 的长度是可知的。这种情况下，当服务端输出完 body 之后，可以考虑使用长连接。能否使用长连接，也是有条件限制的。如果客户端的请求头中的 connection 为 close，则表示客户端需要关掉长连接。如果客户端的请求头中的 connection 为 keepalive，则客户端需要打开长连接，如果客户端的请求中没有 connection 这个头，那

么根据协议,如果是 HTTP 1.0,则默认为 close,如果是 HTTP 1.1,则默认为 keepalive。如果结果为 keepalive, Nginx 在输出完响应体后,会设置当前连接的 keepalive 属性,然后等待客户端下一次请求。当然, Nginx 不可能一直等待下去,如果客户端一直不发数据过来,连接将会被一直占用。所以当 Nginx 设置了 keepalive 等待下一次请求时,同时会设置一个最大等待时间,这个时间是通过选项 keepalive\_timeout 配置的,如果配置为 0,则表示关掉 keepalive,此时, HTTP 版本无论是 1.1 还是 1.0,客户端的 connection 不管是 close 还是 keepalive,都会强制为 close。

如果服务端最后的决定是打开 keepalive,那么在响应的 HTTP 头里面,也会包含 connection 头域,其值是 keepalive,否则就是 close。如果 connection 值为 close,在 Nginx 响应完数据后,会主动关掉连接。对于请求量比较大的 Nginx 来说,关掉 keepalive 会产生比较多的 time-wait 状态 socket。一般来说,当客户端的一次访问,需要多次访问同一个服务器时,打开 keepalive 的优势非常大,如对于图片服务器,通常一个网页会包含很多图片,打开 keepalive 会大量减少 time-wait 状态。

#### 4.2.12 pipeline

在 HTTP 1.1 中,引入了一种新的特性,即 pipeline。pipeline 是流水线作业,可以看作 keepalive 的一种提高,因为 pipeline 也是基于长连接的,目的是利用一个连接做多次请求。如果客户端要提交多个请求,对于 keepalive 来说,那么第二个请求必须要等到第一个请求的响应接收完全后,才能发起,这和 TCP 的停止等待协议是一样的,得到两个响应的时间至少为 2RTT;而对于 pipeline 来说,客户端不必等到第一个请求处理完后,就可以马上发起第二个请求,得到两个响应的时间可能达到 RTT。Nginx 支持 pipeline,但是, Nginx 对 pipeline 中的多个请求的处理不是并行的,依然是一个请求接一个请求地处理,只是在处理第一个请求的时候,客户端就可以发起第二个请求。这样, Nginx 利用 pipeline 减少了处理完一个请求后,等待第二个请求请求头数据的时间。Nginx 的做法很简单: Nginx 在读取数据时,会将读取的数据放到一个 buffer 里面,如果 Nginx 在处理完前一个请求后,发现 buffer 里面还有数据,就认为剩下的数据是下一个请求的开始,接下来处理下一个请求,否则就将连接设置为 keepalive。

### 4.3 小结

本章对 Nginx 的核心架构和关键技术进行了介绍。Nginx 核心是事件驱动的纯异步架构。Nginx 在代码上使用了核心模块和其他模块分开但是共同工作的模块机制。在程序模型上使用了管理进程/工作进程的工作机制,并对管理进程/工程进程机制引起的“惊群”问题和负载均衡的调度做了处理。Nginx 中使用了一些提高系统性能的关键技术,这些技术对于开发高性能服务器是非常有效的,对我们进行 Nginx 配置也会有帮助,我们会知道这些配置项会对哪些模块产生影响,会知道这些取值的背景意义。

## Nginx 的工作流程

详细了解 Nginx 的主要工作流程，可以让我们更好地认识 Nginx，从而更好地使用 Nginx，同时可以让我们更好地掌握 Lua 在 Nginx 中的工作流程。根据对 Nginx 核心技术和架构的了解，我们知道了管理进程 / 工作进程机制，以及 HTTP 核心模块和配置对 Nginx 的重要性，也知道了 Nginx 的框架代码不多，只负责环境管理，本章将介绍这些部分的工作流程。

### 5.1 Nginx 的启动流程

Nginx 启动时分为两部分：

1) 框架程序启动过程：这个阶段会创建各核心模块和非核心模块。

2) 模块启动过程：模块内部完成自己的启动和初始化部分。

Nginx 启动过程如图 5-1 所示。每个过程具体的任务如下：

1) 启动时，Nginx 接受命令行参数，解析出各主要参数。因为 Nginx 的参数主要放在 `nginx.conf` 中，所以最重要的参数是 `nginx.conf` 的路径。

2) 平滑升级是指不重启服务而进行升级，不重启管理进程而启动新版本的 Nginx 程序。旧的管理进程先调用 `fork` 函数创建（即分叉）一个新进程，然后新进程通过 `execve` 系统调用启动新版本管理进程，旧版本管理进程首先设置环境变量，新版本管理进程启动时检查对应环境变量知道是平滑升级，并对通过环境变量传递的旧版本 Nginx 服务监听的句柄做继承处理。

3) 框架通过调用核心模块的 `create_conf` 方法让核心模块创建用于存储对应配置信息的

结构体创建核心模块。一直到第8步，都是基于配置文件对环境和模块进行初始化的。这一步为后面的配置文件解析做好准备工作。

4) 调用配置模块的解析方法，解析 `nginx.conf` 中的配置项。调用对应核心模块的方法将属于各核心模块的配置项保存到核心模块的配置数据结构中。

5) 调用所有核心模块的 `init_conf` 方法，用于让核心模块根据写入内部配置数据结构的数据对模块做处理和初始化。

6) 配置文件中可能配置了缓存文件、库文件、日志文件等，同时包括共享内存，在这一步对这些文件和共享内存进行创建、打开操作。

7) 对于配置了监听端口的模块，按配置开始监听配置的端口。一般 HTTP 模块、stream 模块都会有监听端口。

8) 调用所有模块的 `init_module` 方法，使用配置信息初始化模块。

9) 如果 `nginx.conf` 中配置了 Nginx 为 master 模式（一般都是这种模式），则创建管理进程。

10) 管理进程根据配置的工作进程数，使用一个循环将所需要的工作进程分叉出来。

11) 管理进程根据配置解析过程时解析出来的配置信息，检查相应 path 配置是否配置值，如果配置了，则分叉出独立的 cache manager 进程（这是一个和工作进程并级的进程）。将后端服务器的应答使用文件缓存下来，下次请求时不需要再向后端发送请求，一般用在 `upstream{}` 中。缓存管理器这个进程定期检查缓存状态、查看缓存总量是否超出限制，如果超出则删除最少使用的部分。cache manager 会定期删除过期缓存文件。

12) 同第11步，管理进程根据配置文件查看是否对应的 path 路径配置了路径，如果配置了，则分叉出 cache loader 进程，并且延迟 1 分钟运行。cache loader 进程会遍历配置文件中 `proxy_cache_path` 指定的缓存路径中所有的缓存文件。根据缓存文件的 MD5 编码遍历由 cache manager 进程生成的内存中的缓存文件红黑树和节点结构（`ngx_http_file_cache_node_t`）。如果不存在，则创建新的节点，并将对象的 `rbnode` 和 `queue` 分别插到红黑树和过期队列中；如果存在，则更新相应属性。cache loader 进程实现的是根据缓存文件进行索引重建工作，即在 Nginx 服务重新启动将之前的缓存文件重建索引起来。该进程工作一段时间后将自动退出。

13) 管理进程调用所有模块的 `init_process` 方法。此时，工作进程的启动工作就完成了，工作进程进入自己的消息循环中开始等待处理用户请求。

14) 如果 Nginx 是 single 模式，则直接调用所有模块的 `init_process` 方法，直接以 single 模式启动完毕。单进程模式下，网络端口监听、数据处理等均由管理进程处理，多进程模式下，网络连接和数据处理等由工作进程处理。不管哪种模式，网络端口都是由管理进程创建的。single 模式一般用于调试。

框架代码负责创建核心模块和功能模块，然后由各进程和模块配合起来向用户提供服务。下面将分别解析主要进程和服务的工作流程。



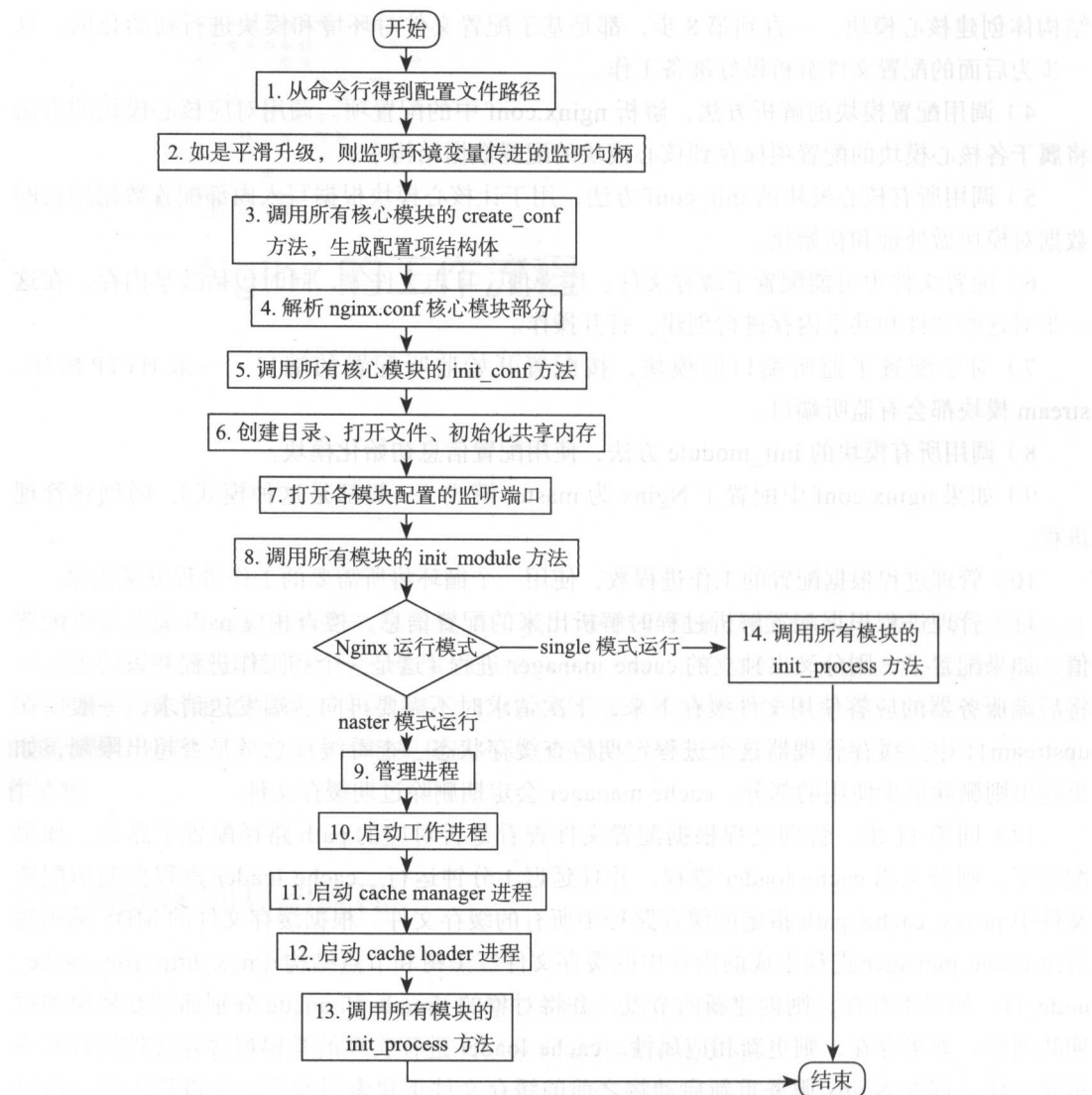


图 5-1 Nginx 启动流程

## 5.2 管理进程的工作流程

管理进程的工作比较简单，它只是管理工作等子进程，实现重启服务、平滑升级、更换日志文件、动态重新装载配置文件等操作，不需要处理网络任务。

用户通过信号操作管理进程。管理进程内部设置了信号，并注册了相应的 handler，信号发生时，会调用相应的 handler 处理对应的请求。我们通过命令行参数操作 Nginx，内部的实现是通过管理进程接收用户输入的信号并处理实现的。

管理进程信号定义如表 5-1 所示。

表 5-1 管理进程信号定义

信号	master 定义的变量	意义
QUIT	ngx_quit	“优雅”地关闭整个服务
TERM 或 INT	ngx_terminate	强制关闭整个服务
USR1	ngx_reopen	重新打开服务中的所有文件
WINCH	ngx_noaccept	所有子进程不再接受处理新的连接, 实际相当于对所有的子进程发送 QUIT 信号量
USR2	ngx_change_binary	平滑升级到最新版本的 Nginx
HUP	ngx_reconfigure	重读配置文件并使服务对新配置项生效
CHLD	ngx_reap	有子进程意外结束, 需要监控所有子进程

管理进程收到信号后, 会设置内部定义的 7 个变量, 在主循环中, 根据这 7 个变量决定 ngx\_master\_process\_cycle 方法如何执行, 实现内部调用, 完成预定的逻辑。

管理进程通过 fork 命令创建工作进程, 并将工作进程号保存到内部进程变量表 (ngx\_processes) 中, 管理进程依靠信号改变表中进程的状态。当子进程意外退出时, 管理进程作为父进程, 会收到 Linux 内核发来的 CHLD 信号, 根据进程 ID 修改内存中的进程状态。

Nginx 设计管理进程 / 工作进程机制的目的是让管理进程监控和管理工作进程, 当工作进程意外终止时, 管理进程需要启动新的工作进程接替终止进程, 实现系统的连续运行能力, 提供高可靠性。下面介绍管理进程的工作循环机制, 从中也可以看到其如何创建新工作进程的机制如图 5-2 所示。

管理进程的工作流程如图 5-2 所示。

管理进程的工作循环根据 8 个状态位 (7 个信号对应状态位与 1 个 no\_noaccept 状态位) 执行不同的代码路径。每当一个循环执行完毕后进程便被挂起, 直到有新的信号才会被激活并继续执行。

1) 如果 ngx\_reap 为 0, 则执行第 2 步; 如果 ngx\_reap 为 1, 表示要监控所有的子进程, 检查每个子进程的状态, 非正常退出的子进程会重新启动。本阶段会返回一个 live 状态, 0 表示所有子进程均已经正常退出, 1 表示所有子进程均正常。

2) 当 live 为 0, 同时 ngx\_terminate 为 1 或者 ngx\_quit 为 1 时, 退出管理进程, 并首先删除保存管理进程号的 pid 文件。

3) 调用所有模块的 exit\_master 方法。

4) 关闭所有监听端口。

5) 销毁内存池, 退出管理进程。

6) 如果 ngx\_terminate 为 1, 则向所有子进程发送 TERM 信号, 通知子进程执行强行退出流程; 然后跳转到第 1 步挂起进程。

7) 如果 ngx\_quit 为 1, 表示需要“优雅”地退出服务, 向所有子进程发送 QUIT 信号; 否则判断 ngx\_reconfigure 标志。



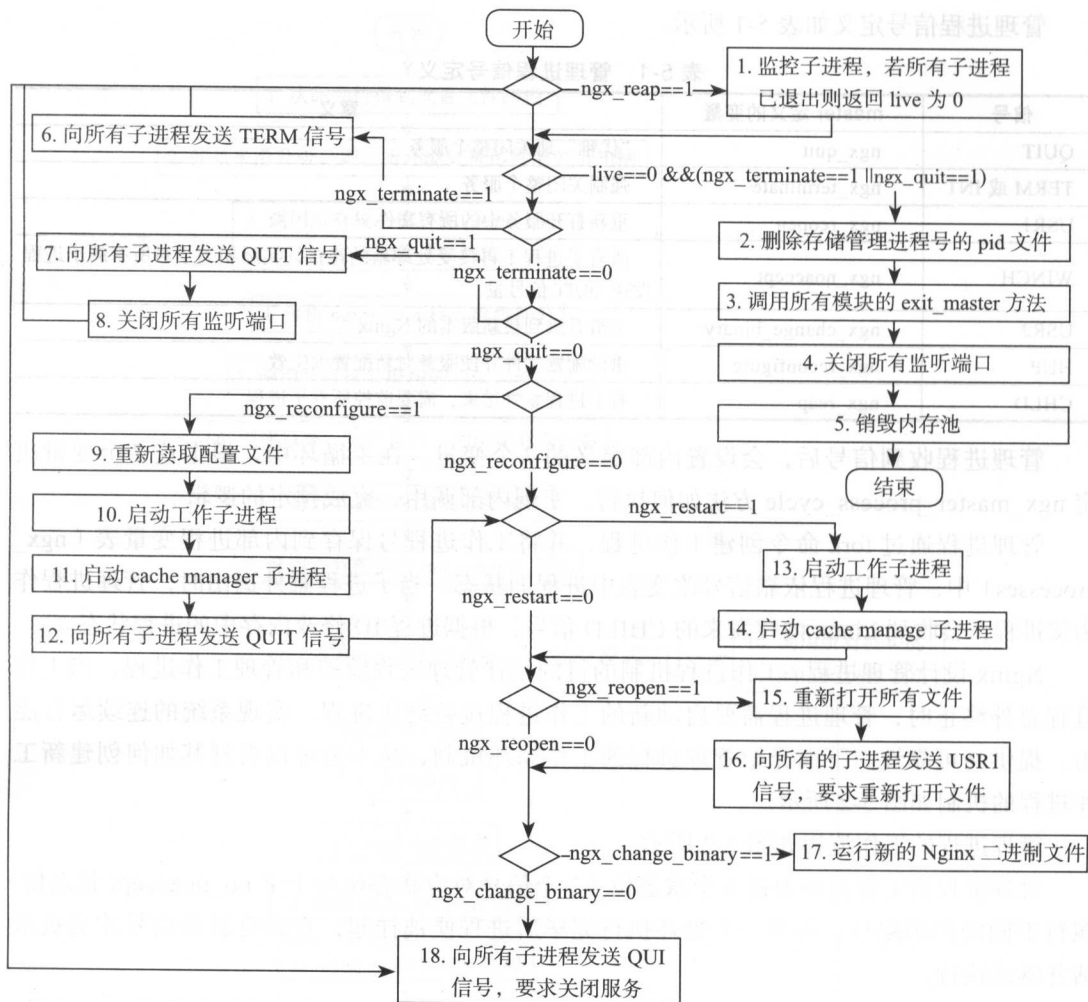


图 5-2 管理进程的工作流程

8) 继续执行 `ngx_quit` 为 1 的分支操作, 关闭所有监听端口, 然后跳转到第 1 步挂起进程。

9) 当 `ngx_reconfigure` 为 1 时, 重新读取配置文件。在这个过程中, 管理进程首先重新初始化配置结构体, 用来读取新的配置文件, 再创建新的工作进程, 然后销毁旧的工作进程。

10) 使用新配置创建新的进程。

11) 根据缓存模块中的配置, 决定是否创建新的 `cache manager` 或 `cache loader` 进程, 同时将内部 `live` 标志置 1。

12) 向旧的子进程发送 `QUIT` 信号, 旧子进程“优雅”地退出。

13) 启动子进程。

14) 根据是否有缓存文件决定启动 `cache manager` 或 `cache loader`, 同时将 `live` 置 1。

15) 如果 `ngx_reopen` 标志为 1, 重新打开所有文件。

- 16) 向所有子进程发送 USR1 信号, 要求子进程重新打开所有文件。
- 17) 检查 ngx\_change\_binary 标志位, 为 1 表示需要平滑升级, 则创建新的工作进程。
- 18) 如果 ngx\_noaccept 标志位为 1, 则向所有子进程发送 QUIT 信号, 让子进程“优雅”地退出。如果 ngx\_noaccept 为 0 则跳转到第 1 步。

### 5.3 工作进程的工作流程

Nginx 的业务处理是在工作进程中完成的, 但实际上是通过工作进程协调各模块组件完成任务的。工作进程由管理进程管理, 它们之间的工作机制是通过信号实现的, 工作进程中有一个专用的方法处理信号, 工作进程关注 4 个信号, 对应到 4 个全局变量, 分别为 ngx\_terminate、ngx\_quit、ngx\_exiting、ngx\_reopen。

除了 ngx\_exiting 标志位 (退出时作为标志位使用), 其他 3 个标志位均由对应信号设置:

- ngx\_terminate: TERM 信号, 对应强制退出操作。
- ngx\_reopen: USR1 信号, 重新打开文件。
- ngx\_quit: QUIT 信号, “优雅”地退出。

工作进程的工作流程如图 5-3 所示。

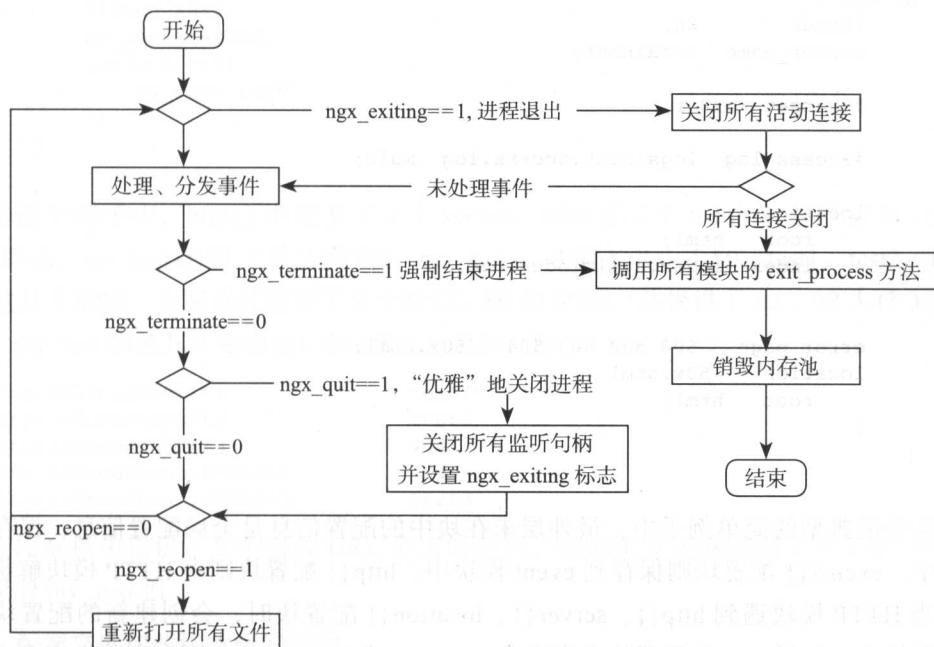


图 5-3 工作进程的工作流程

工作进程处理 4 个信号量, 作为主流程, 否则工作循环处理网络 event 事件, 处理 HTTP 业务流程。

## 5.4 配置加载流程

Nginx 服务是通过 nginx.conf 配置文件实现的, Nginx 是多模块架构, 在框架启动流程中, 每个模块都会为自己创建一个配置信息数据结构, 而框架又会调用模块 `init_conf` 接口, 将配置项加载到模块一级。所有配置项中的配置以配置块为单位, 而配置块又是与内部模块对应的。配置项配置信息保存在模块中。

下面是一个典型的 nginx.conf 配置文件。

```
#user nobody;
worker_processes 1;

events {
    worker_connections 1024;
}

http {
    include mime.types;
    default_type application/octet-stream;

    sendfile on;
    keepalive_timeout 65;

    server {
        listen 80;
        server_name localhost;

        #charset koi8-r;

        #access_log logs/host.access.log main;

        location / {
            root html;
            index index.html index.htm;
        }

        error_page 500 502 503 504 /50x.html;
        location = /50x.html {
            root html;
        }
    }
}
```

在这个最典型的简单例子中, 最外层未在块中的配置信息是全局配置信息, 保存在框架变量中。events{} 配置块则保存到 event 模块中。http{} 配置块则由 HTTP 模块解析并保存, 而当 HTTP 模块遇到 http{}、server{}、location{} 配置块时, 会创建新的配置块数据结构保存信息, 如果 http{} 配置块中有多个 server{} 或 location{}, 则内存将会有多个配置数据结构。

框架程序在解析配置文件时, 通过 3 个 HTTP 模块的回调函数将配置信息传给 HTTP 模块: `create_main_conf`、`create_svr_conf`、`create_loc_conf`, 3 个函数分别对应 http{}、server{}、

location{} 配置块。create\_main\_conf 只会被调用一次，而 create\_svr\_conf 和 create\_loc\_conf 则可能会被调用多次，这取决于配置情况。HTTP 模块会多次生成配置数据结构保存配置信息。

取决于 Nginx 的这种配置实现，Nginx 其实就有了一些特性，可以在不同的位置配置多个 location，我们来看下面这个配置例子：

```
http{
    my_test hello;

    server{
        listen 80;
        my_test is80;

        location /t1{
            my_test test1;
        }

        location /t2{
            my_test test2;
        }
    }

    server{
        listen 8080;
        my_test is8080;
        location /t3{
            my_test test3;
        };
    }
}
```

在这个例子中，http{} 中定义了 2 个 server，同时在 3 个 location 中重新赋值了 my\_test，那么，my\_test 的哪个值生效呢？my\_test 一共有 hello、is80、test1、test2、is8080、test3 这几个取值。本配置共监听了 2 个端口：80 和 8080，共提供了 /t1、/t2、/t3 3 个 URL 请求。my\_test 的值其实是由访问的 URL 决定的，即

http://localhost/t1	test1
http://localhost/t2	test2
http://localhost/t3	404
http://localhost:8080/t1	404
http://localhost:8080/t3	test3

上面解析中提供了 2 个错误 URL 请求的示例，按照配置正确访问 3 个 URL 则可以得到对应的 my\_test 值。

### 1. 配置文件加载和解析的过程

配置文件加载和解析的过程参见图 5-1 的 Nginx 启动流程。主要过程如下：

- 1) 从 Nginx 命令行得到配置文件路径。
- 2) 调用所有核心模块的 create\_conf 方法，生成配置项结构体。

3) 框架代码解析 `nginx.conf` 核心模块部分。

4) 调用所有核心模块的 `init_conf` 方法，解析对应的配置块。

## 2. HTTP 配置块解析过程

核心模块将配置主业务分解后，下一阶段就是核心模块的配置加载和解析过程了，下面以 HTTP 模块为例解析具体解析过程。

图 5-4 描述了在主架构内 HTTP 配置块解析过程。

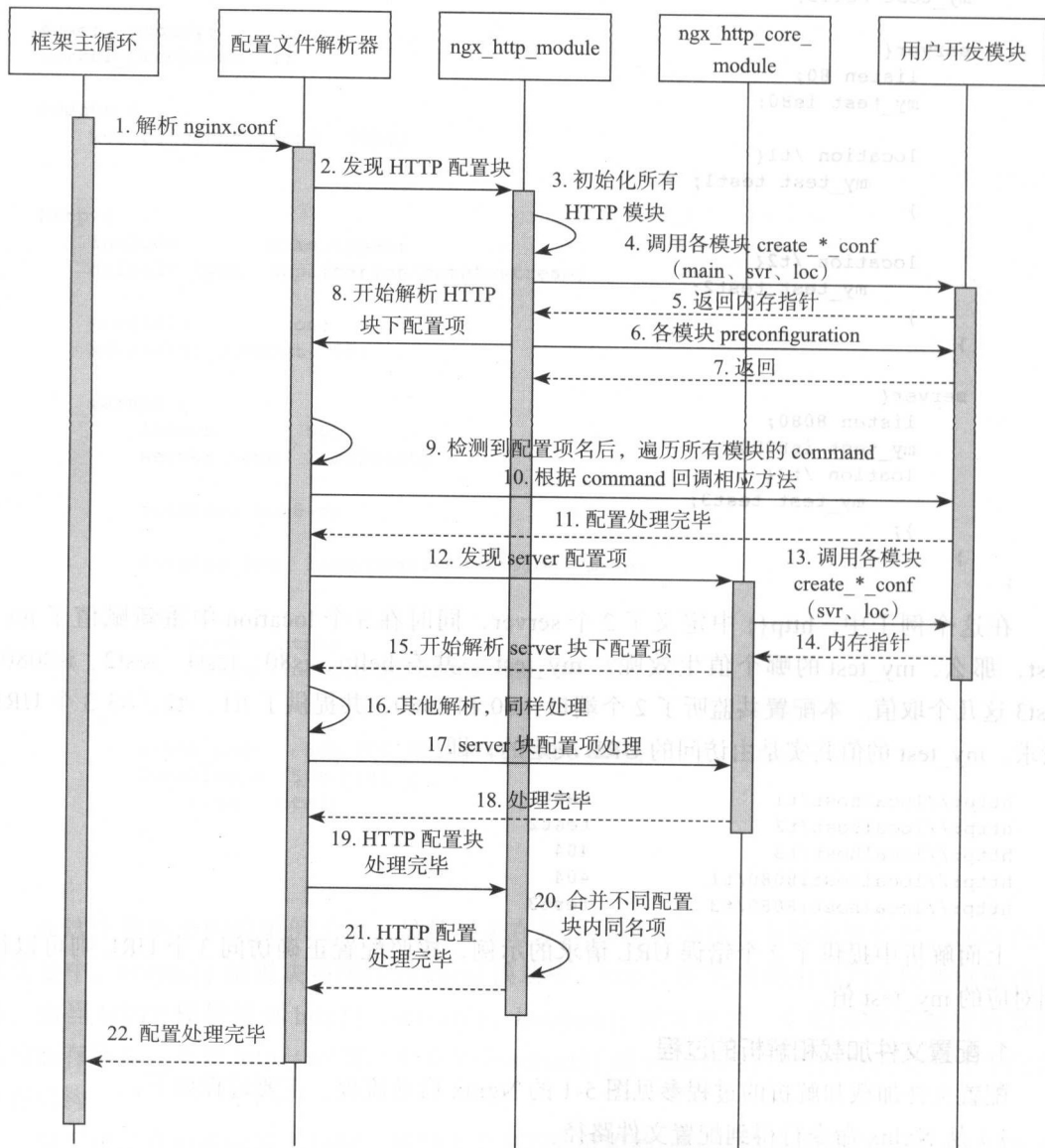


图 5-4 HTTP 配置项解析过程

整个过程比较简洁易懂，不再重复描述。程序主架构指的是 Nginx 的框架代码，在系统启动时调用配置文件解析器解析 `nginx.conf` 文件。

Nginx 系统配置文件解析还有一个渠道，就是管理进程提供的重新载入命令，一般是“`nginx -s reload`”，在管理进程收到这个命令或消息时，会初始化存放配置信息的数据结构，然后使用新配置启动新的工作进程，然后杀掉旧的工作进程。这个过程也会触发上面的流程再次执行。

## 5.5 HTTP 框架初始化流程

HTTP 模块是 Nginx 中的核心模块，作为一个 Web 服务器而言，Nginx 是工作在 HTTP 协议之上的，所以 HTTP 业务的处理是最重要的工作。处理 HTTP 协议，是基于网络层的，HTTP 是 TCP 协议。HTTP 部分是由核心模块 `ngx_http_module`、`ngx_http_core_module`、`ngx_http_upstream_module` 模块组成的。

HTTP 模块的主要职责如下：

- 1) 调用 HTTP 框架提供的接口发送 HTTP 应答。
- 2) 分解出若干子请求进行复杂业务的处理，子请求也是全异步非阻塞式处理。
- 3) 将一个 HTTP 请求分为顺序化的多个处理阶段，依次进行流水线式处理。如果前一个流水线阶段决定中断处理，则后续流水线处理部分将不会被调用到。
- 4) 异步接收 HTTP 请求中的包体，可以将数据缓存到文件中。
- 5) 异步访问第三方服务。
- 6) 处理已经解析完的 HTTP 请求。
- 7) 解析 `nginx.conf` 中属于自己的配置项。在不同的 `http{}、server{}、location{} 下的配置项，都需要正确解析。`

HTTP 框架代码需要具备下述功能：

- 1) 向 HTTP 模块提供磁盘 I/O 功能和网络 I/O 功能。
- 2) 提供 upstream 机制使 HTTP 模块可以访问第三方服务。
- 3) 使用 event 模块，以处理所有网络事件。
- 4) 提供子请求机制实现子请求功能。
- 5) 辨别接收到的 TCP 流是否是完整的 HTTP 报文。
- 6) 根据请求中的 URI 和头域，根据配置文件将请求分发到对应的模块，调用模块注册的函数处理请求。
- 7) 解析和处理 `nginx.conf` 文件，解析 `http{} 配置块，解析 http{} 下的 server{}、location{} 等子配置块，可以处理同名配置项出现在不同配置块中的情况。`

图 5-5 描述了 HTTP 框架初始化流程。

主要流程简述如下：

1) 初始化 `ngx_module` 数据中所有 HTTP 模块的 `ctx_index` 字段, 从 0 开始递增。这个索引就是请求响应时调用的顺序, 而这个顺序最终是由编译 Nginx 时的模块顺序决定的, 同时初始化存放配置信息的 `ngx_http_conf_ctx_t` 数据结构。

2) 依次调用所有 HTTP 模块的 `create_main_conf` 方法, 产生的配置结构体指针按照各模块的 `ctx_index` 字段顺序放入 `ngx_http_conf_ctx_t` 的 `main_conf` 数组。

3) 依次调用所有 HTTP 模块的 `create_svr_conf` 方法, 产生的配置结构体指针按照各模块的 `ctx_index` 字段顺序放入 `ngx_http_conf_ctx_t` 的 `svr_conf` 数组。

4) 依次调用所有 HTTP 模块的 `create_loc_conf` 方法, 产生的配置结构体指针按照各模块的 `ctx_index` 字段顺序放入 `ngx_http_conf_ctx_t` 的 `loc_conf` 数组。

5) 依次调用所有模块的 `preconfiguration` 方法, `preconfiguration` 回调函数完成了对应模块的预处理操作, 其主要工作是创建模块用到的变量。

6) 调用所有 HTTP 模块的 `init_conf` 方法, 告诉模块配置解析完成。

7) 合并配置项。

8) Nginx 将 HTTP 处理过程划分成了 11 个阶段, 使多个模块可以介入到不同的阶段进行流水线式操作, 充分发挥模块式架构的优势, 并实现请求过程异步化。其中有 7 个阶段是允许用户介入的: `NGX_HTTP_POST_READ_PHASE`、`NGX_HTTP_SERVER_REWRITE_PHASE`、`NGX_HTTP_REWRITE_PHASE`、`NGX_HTTP_PREACCESS_PHASE`、`NGX_HTTP_ACCESS_PHASE`、`NGX_HTTP_CONTENT_PHASE`、`NGX_HTTP_LOG_PHASE`。调用 `ngx_http_init_phases` 方法初始化这 7 个动态数组, 数据保存在 `phases` 数组中。

9) 依次调用所有 HTTP 模块的 `postconfiguration` 方法, 使 HTTP 模块可以处理 HTTP 阶段, 将 HTTP 模块的 `ngx_http_handler_pt` 处理方法添加到 HTTP 阶段中。

10) 构建虚拟主机的查找散列表。虚拟主机配置在 `server{}` 中, 为了提高请求时查找的速度, 使用散列表对主机 `server name` 进行了索引。

11) 建立 `server` 与监听端口间的关联, 同时设置新连接的回调方法。

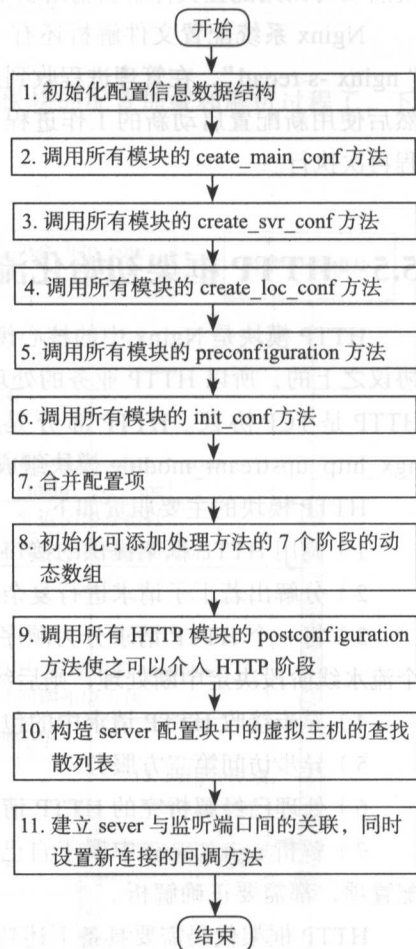


图 5-5 HTTP 框架初始化流程



## 5.6 HTTP 模块调用流程

HTTP 框架执行流程涉及底层事件模型，全异步的工作方式比较复杂，对于 Nginx 下的 Lua 开发指导意义不大，所以本书不会展开描述这部分知识，有需要的读者请自行研究。本节对 HTTP 模块间的调用流程进行简单介绍，这个流程将向我们展示配置的模块间的关系和分工，便于对 HTTP 模块有一个整体了解。

图 5-6 描述了一个简略的 HTTP 模块调用流程，精简了很多过程，去除了异步的处理机制。

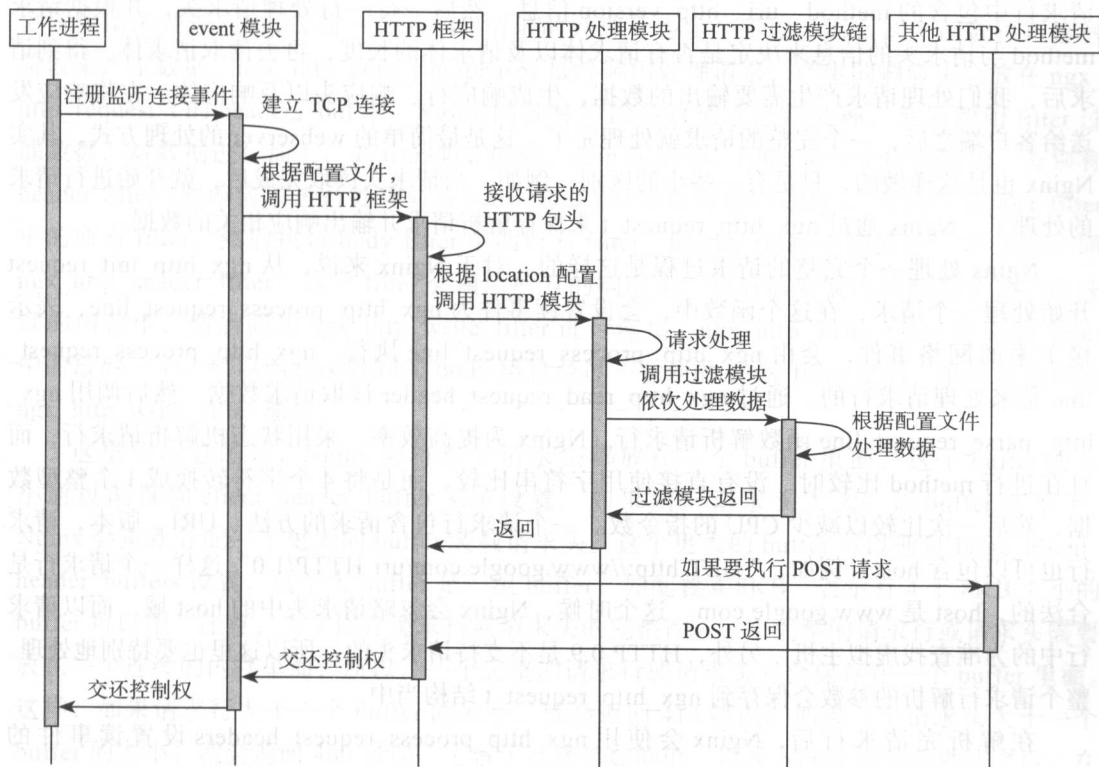


图 5-6 HTTP 模块调用流程

工作进程在主循环调用事件模型，检测网络事件，当有新连接请求时，则建立 TCP 连接，然后根据 `nginx.conf` 配置，将请求交由 HTTP 框架处理。框架首先尝试接收 HTTP 头部，接收到完整 HTTP 头部后，将请求分发到具体的 HTTP 模块处理。通常根据 URI 和 `nginx.conf` 里的 `location` 匹配程度决定分发策略。请求处理结束时，通常都向客户端发送响应，这时一般自动依次调用所有的 HTTP 过滤模块，每个模块根据配置文件中定义的策略决定自己的行为，如可调用 `gzip` 模块根据 `nginx.conf` 中“`gzip on|off;`”决定是否将响应压缩。如果设置了子请求调用，在返回前还会执行异步的子请求调用。

## 5.7 HTTP 请求处理流程

Nginx 中的 request 指的是 HTTP 请求，在 Nginx 中对应的数据结构是 `ngx_http_request_t`。`ngx_http_request_t` 是对 HTTP 请求的封装。根据 HTTP 规范，一个 HTTP 请求包含请求行、请求头、请求体、响应行、响应头、响应体。

HTTP 请求是典型的请求-响应类型的网络协议，而 HTTP 是文件协议，所以在分析请求行与请求头，以及输出响应行与响应头时，往往是一行一行地处理。如果自己编写一个 HTTP 服务器，通常在一个连接建立好后，客户端会发送请求过来。读取一行数据，分析请求行中包含的 `method`、`uri`、`http_version` 信息。然后一行一行处理请求头，并根据请求 `method` 与请求头的信息来决定是否有请求体以及请求体的长度，再去读取请求体。得到请求后，我们处理请求产生需要输出的数据，生成响应行、响应头以及响应体。在将响应发送给客户端之后，一个完整的请求就处理完了。这是最简单的 `webserver` 的处理方式，其实 Nginx 也是这样做的，只是有一些小的区别，例如，当请求头读取完成后，就开始进行请求的处理了。Nginx 通过 `ngx_http_request_t` 来保存解析请求并输出响应相关的数据。

Nginx 处理一个完整的请求过程是这样的。对于 Nginx 来说，从 `ngx_http_init_request` 开始处理一个请求，在这个函数中，会设置读事件为 `ngx_http_process_request_line`，表示接下来的网络事件，会由 `ngx_http_process_request_line` 执行。`ngx_http_process_request_line` 是来处理请求行的。通过 `ngx_http_read_request_header` 读取请求数据，然后调用 `ngx_http_parse_request_line` 函数解析请求行。Nginx 为提高效率，采用状态机解析请求行，而且在进行 `method` 比较时，没有直接使用字符串比较，而是将 4 个字符转换成 1 个整型数据，然后一次比较以减少 CPU 的指令数。一个请求行包含请求的方法、URI、版本，请求行也可以包含 `host`。例如，“GET `http://www.google.com/uri` HTTP/1.0” 这样一个请求行是合法的，`host` 是 `www.google.com`。这个时候，Nginx 会忽略请求头中的 `host` 域，而以请求行中的为准查找虚拟主机。另外，HTTP 0.9 是不支持请求头的，所以这里也要特别地处理。整个请求行解析的参数会保存到 `ngx_http_request_t` 结构当中。

在解析完请求行后，Nginx 会使用 `ngx_http_process_request_headers` 设置读事件的 handler，然后后续的请求就在 `ngx_http_process_request_headers` 中进行读取与解析。`ngx_http_process_request_headers` 函数用来读取请求头，跟请求行一样，还是调用 `ngx_http_read_request_header` 读取请求头，调用 `ngx_http_parse_header_line` 解析一行请求头，解析到的请求头会保存到 `ngx_http_request_t` 的域 `headers_in` 中，`headers_in` 是一个链表结构，保存所有的请求头。而 HTTP 中有些请求是需要特别处理的，这些请求头与请求处理函数存放在一个映射表里面，即 `ngx_http_headers_in`。在初始化时，会生成一个 hash 表，每当解析一个请求头后，就会先在 hash 表中查找，如果找到，则调用相应的处理函数处理这个请求头。例如，`host` 头的处理函数是 `ngx_http_process_host`。

当 Nginx 解析到两个回车换行符时，就表示请求头已经结束，此时会调用 `ngx_http_`

`process_request` 处理请求。`ngx_http_process_request` 会设置当前连接的读写事件处理函数为 `ngx_http_request_handler`，然后调用 `ngx_http_handler` 真正开始处理一个完整的 HTTP 请求。读写事件处理函数 `ngx_http_request_handler` 在代码中会根据当前事件是读事件还是写事件，分别调用 `ngx_http_request_t` 中的 `read_event_handler` 或者 `write_event_handler`。由于此时，请求头已经读取完成，因为 Nginx 的做法是先不读取请求 body，所以这里面设置 `read_event_handler` 为 `ngx_http_block_reading`，即不读取数据。真正开始处理数据，是在 `ngx_http_handler` 这个函数里面，这个函数会设置 `write_event_handler` 为 `ngx_http_core_run_phases`，并执行 `ngx_http_core_run_phases` 函数。`ngx_http_core_run_phases` 函数将执行多阶段请求处理，Nginx 将一个 HTTP 请求的处理分为多个阶段，那么这个函数执行这些阶段来产生数据。`ngx_http_core_run_phases` 最终调用处理请求，产生的响应头会放在 `ngx_http_request_t` 的 `headers_out` 中。Nginx 的各种阶段会对请求进行处理，最后调用 `filter` 过滤数据，对数据进行加工，如 `truncked` 传输、`gzip` 压缩等。`filter` 是一个链表结构，分别有 `header filter`（对响应头进行处理）与 `body filter`（对响应体进行处理），先执行 `header filter` 中的所有 `filter`，然后执行 `body filter` 中的所有 `filter`。`header filter` 中的最后一个 `filter`，即 `ngx_http_header_filter`，这个 `filter` 会遍历所有的响应头，最后需要输出的响应头在一个连续的内存中，然后调用 `ngx_http_write_filter` 进行输出。`ngx_http_write_filter` 是 `body filter` 中的最后一个，所以 Nginx 收到的 `body` 信息经过一系列的 `body filter` 之后，最后也会调用 `ngx_http_write_filter` 输出。

这里要注意的是，Nginx 会将整个请求头都放在一个 `buffer` 里面，这个 `buffer` 的大小通过配置项 `client_header_buffer_size` 设置。如果用户请求头太大，该 `buffer` 装不下，Nginx 会重新分配一个更大的 `buffer` 装载请求头。这个更大的 `buffer` 可以通过 `large_client_header_buffers` 设置，这个大 `buffer` 是一组 `buffer`，如配置 4 8KB，表示有 4 个 8KB 大小的 `buffer` 可以用。注意，为了保存请求行或请求头的完整性，一个完整的请求行或请求头需要放在一个连续的内存里面，所以，一个完整的请求行或请求头只会保存在一个 `buffer` 里面。这样，如果请求行大于一个 `buffer` 的大小，就会返回 414 错误，如果一个请求头大于一个 `buffer` 的大小，就会返回 400 错误。了解了这些参数的值，以及 Nginx 的实际做法之后，在具体的应用场景下，就可以根据实际的需求调整这些参数。

## 5.8 小结

本章介绍了 Nginx 的启动流程、管理进程与工作进程的工作流程、配置项加载流程、HTTP 模块初始化流程、HTTP 模块调用流程及 HTTP 请求处理流程。通过对流程的介绍，可以让我们对各个阶段工作的上下文比较了解，更容易理解 API 的工作条件，有助于我们开发更高性能的 Lua 应用。



## 第二部分 Part 2

# Lua 脚本语言

- 第6章 Lua 教程
- 第7章 Lua 通用库

章 8 是

野姓 su

脚本 su 1.0

卦卦 20 su 1.1.0

野姓 su 1.0

# Lua 教程

Lua 是一种轻量、小巧的脚本语言，用标准 C 语言编写并以源代码形式开放。其设计目的是嵌入应用程序中，从而为应用程序提供灵活的扩展和定制功能。目前 Lua 大量应用于 Nginx、嵌入式设备、游戏逻辑开发等方面。

在 Nginx 中，Lua 中的例程机制可以很好地和 Nginx 的全异步、非阻塞的多阶段处理机制结合，使开发者使用同步的模式，开发全异步的应用，而不用考虑异步的处理机制。

本章介绍 Lua 的基础语法知识。

## 6.1 Lua 基础

推荐读者在自己的计算机上安装一个 Lua 开发环境，因为 Lua 是解释型脚本语言，可以一行一行地调试，便于快速理解。

### 6.1.1 Lua 的特性

相较于其他语言、其他脚本语言，Lua 有其自身的特点。

- 轻量级：Lua 用标准 C 语言编写并以源代码形式开放，编译后仅仅一百余千字节，可以很方便地嵌入其他程序中。
- 可扩展：Lua 提供了非常易于使用的扩展接口和机制，由宿主语言（通常是 C 或 C++）提供功能，Lua 可以使用它们，就像内置的功能一样。
- 其他特性：
  - ✓ 支持面向过程编程和函数式编程。

- ✓ 自动内存管理：只提供了一种通用类型的表（table），用它可以实现数组、hash 表、集合、对象。
- ✓ 语言内置模式匹配：①闭包（closure）；②函数也可以看作一个值；③提供多线程（协同进程，简称协程，并非操作系统所支持的线程）支持。
- ✓ 通过闭包和 table 可以很方便地支持面向对象编程所需要的一些关键机制，如数据抽象、虚函数、继承和重载等。

### 6.1.2 Lua 的应用场景

Lua 在不同的系统中得到了大量应用，常见的应用场景如下。

- 游戏开发：在游戏开发中，游戏的逻辑因为需要经常修改，所以游戏设计了一个模式，将游戏的操作收集和显示放在客户端，逻辑放在后端服务器上。在服务器上使用 Lua 脚本实现，这样当游戏规则修改或游戏规模变大而要修改逻辑时可以用不用升级游戏客户端，而且在服务端修改起来也很简单。
- 独立应用脚本：Lua 的语法类似于 C 和 C++，支持常用的系统库，可以编写简单的独立小程序，如一些纯数学计算应用、类似于 Shell 的复杂脚本。
- Web 应用脚本：使用 Lua 实现 CGI 应用。CGI 接口简单，使用 CGI 的应用程序对 HTTP 请求参数进行处理，将参数与数据库建立关系，然后返回格式化的数据。可以在这种场景下使用 Lua 进行字符串处理和数据格式化操作。
- 扩展和数据库插件，如使用在 MySQL 的 Proxy 里。Proxy 的工作是根据配置做调写分离调度，同时实现数据同步操作。是纯粹的业务逻辑，与磁盘的高速交互由对应的引擎处理。这种纯逻辑性工作适合用 Lua 开发，开发速度快，修改灵活。
- 安全系统，如入侵检测系统：在 WAF 里，以及其他的应用层协议检测机制中，Lua 可以用于做入侵规则判断。其相比于 C 语言的好处是规则可以动态加载，可以及时修改。不同的检测规则编写在不同的 Lua 文件中，根据配置调用对应的 Lua 文件，可以实现使用不同规则的机制。

### 6.1.3 安装 Lua 环境

为了运行 Lua 代码，可以在自己的开发环境上安装 Lua，这比在 Nginx 上直接调试 Lua 要方便很多，所以推荐先使用本文讲解的工具学习 Lua 语法，之后再在 Nginx 上开始 Web 开发。

#### 1. 在 Linux 系统上安装 Lua

在 Linux 上安装 Lua 非常简单，只需要下载源码包并在终端解压、编译即可，本文使用了 5.3.0 版本进行安装：

```
curl -R -O http://www.lua.org/ftp/lua-5.3.0.tar.gz
```

```
tar xzf lua-5.3.0.tar.gz
cd lua-5.3.0
make linux test
make install
```

## 2. 在 Mac OS X 系统上安装 Lua

在 Mac OS X 系统执行下面命令进行安装：

```
curl -R -O http://www.lua.org/ftp/lua-5.3.0.tar.gz
tar xzf lua-5.3.0.tar.gz
cd lua-5.3.0
make macosx test
make install
```

接下来创建一个 HelloWorld.lua 文件，代码如下：

```
print("Hello World!");
```

执行以下命令：

```
$ lua HelloWorld.lua
```

输出结果为

```
Hello World!
```

## 3. 在 Windows 系统上安装 Lua

在 Windows 下可以使用 SciTE 的集成开发环境（Integrated Development Environment, IDE）来执行 Lua 程序，下载地址为 [http://static.runoob.com/download/LuaForWindows\\_v5.1.4-46.exe](http://static.runoob.com/download/LuaForWindows_v5.1.4-46.exe)。

GitHub 下载地址：<https://github.com/rjpccomputing/luaforwindows/releases>。

Google Code 下载地址：<https://code.google.com/p/luaforwindows/downloads/list>。

双击安装后即可在该环境下编写 Lua 程序并运行。

也可以使用 Lua 官方推荐的方法使用 LuaDist：<http://luadist.org/>。

## 6.2 Lua 基本语法

Lua 与 C/C++ 语言非常相似，大量使用了符号简化逻辑描述，所以整体上比较清晰、简洁。条件语句、循环语句、函数调用等与 C/C++ 基本一致。

### 6.2.1 第一个 Lua 程序

Lua 有交互式和脚本式两种编程方式，下面分别讲解。

#### 1. 交互式编程

Lua 提供了交互式编程模式，可以在命令行中输入程序并立即查看效果。

Lua 交互式编程模式可以通过命令 `lua -i` 或 `lua` 来启用：



```
$ lua -i
$ Lua 5.3.0 Copyright (C) 1994-2015 Lua.org, PUC-Rio
>
```

在命令行中, 输入以下命令:

```
> print("Hello World! ")
```

按下回车键, 输出结果如下:

```
> print("Hello World! ")
Hello World!
>
```

## 2. 脚本式编程

可以将 Lua 代码保存到一个以 lua 为扩展名的文件中并执行, 该模式称为脚本式编程。

例如, 将如下代码存储在名为 hello.lua 的脚本文件中:

```
print("Hello World! ")
```

使用 Lua 执行以上脚本, 输出结果为

```
$ lua test.lua
Hello World!
```

也可以将代码修改为如下形式来执行脚本 (在开头添加 #!/usr/local/bin/lua):

```
#!/usr/local/bin/lua
print("Hello World! ")
```

以上代码中, 指定了 Lua 解释器为 /usr/local/bin。加上 # 号标记解释器会忽略它。接下来为脚本添加可执行权限, 并执行:

```
chmod 777 test.lua
./test.lua
Hello World!
```

### 6.2.2 注释

与其他语言的注释语法一样, Lua 支持两种注释方法。

#### 1. 单行注释

单行注释格式:

```
-- 注释
```

#### 2. 多行注释

多行注释格式:

```
--[[
多行注释
多行注释
--]]
```

### 6.2.3 标识符

Lua 标识符用于定义一个变量、函数或其他用户定义的项。标识符以一个字母 (A ~ Z 或 a ~ z) 或下划线 ( \_ ) 开头后加上 0 个或多个字母、下划线、数字 (0 ~ 9)。

最好不要使用下划线加大写字母的标识符, 因为 Lua 的保留字也是这样定义的。

Lua 不允许使用特殊字符 (如 @、\$、和 %) 来定义标识符。Lua 是一个区分大小写的编程语言, 因此在 Lua 中 test 与 Test 是两个不同的标识符。

以下列出了一些正确的标识符: mohd、zara、abc、move\_name、a\_123、myname50、\_temp、j、a23b9、retVal。

### 6.2.4 关键词

表 6-1 列出了 Lua 保留关键字。保留关键字不能作为常量、变量或其他用户自定义标识符。

一般约定, 以下划线开头连接一串大写字母的名字 (如 \_VERSION) 被保留用于 Lua 内部全局变量。

表 6-1 Lua 保留关键字

and	break	false	local
elseif	end	in	repeat
function	if	or	until
nil	not	true	
return	then	else	
while	do	for	

### 6.2.5 全局变量

默认情况下, 变量总是被认为是全局的。

全局变量不需要声明, 给一个变量赋值后即创建了这个全局变量, 访问一个没有初始化的全局变量也不会出错, 只不过得到的结果是 nil。

```
>print(b)
nil
> b=10
>print(b)
10
>
```

如果想删除一个全局变量, 只需要将变量赋值为 nil。

```
b = nil
print(b)      --> nil
```

这样变量 b 就好像从没被使用过一样。换句话说, 当且仅当一个变量不等于 nil 时, 这个变量即存在。

## 6.3 Lua 的数据类型

Lua 是动态类型语言, 变量不需要类型定义, 只需要为变量赋值。值可以存储在变量中, 作为参数传递或作为结果返回。

Lua 中有 8 个基本类型, 分别为 nil、boolean、number、string、userdata、function、

thread 和 table。

Lua 基本数据类型如表 6-2 所示。

表 6-2 Lua 基本数据类型

数据类型	描述
nil	无效值，在条件表达式中相当于 false。只有 nil 这一个值
boolean	包含两个值：false 和 true
number	表示双精度类型的实浮点数
string	字符串由一对双引号或单引号表示
function	由 C 或 Lua 编写的函数
userdata	表示任意存储在变量中的 C 数据结构
thread	执行的独立线程，用于执行协同程序
table	Lua 中的表实际上是一个“关联数组”，索引可以是数字或字符串。 Lua 中，表通过构造表达式完成。最简单的表达式是 {}，用来创建一个空表

可以使用 type 函数测试给定变量或者值的类型：

```
print(type("Hello world")) --> string
print(type(10.4*3))        --> number
print(type(print))         --> function
print(type(type))          --> function
print(type(true))          --> boolean
print(type(nil))           --> nil
print(type(type(X)))        --> string
```

### 1. nil (空)

nil 类型表示没有任何有效值，它只有一个值：nil。例如，打印一个没有赋值的变量，便会输出 nil 值：

```
>print(type(a))
nil
>
```

对于全局变量和 table，nil 还有“删除”作用，给全局变量或者 table 表里的变量赋 nil 值，等同于把它们删掉，执行下面代码：

```
tab1 = { key1 = "val1", key2 = "val2", "val3" }
for k, v in pairs(tab1) do
    print(k .. " - " .. v)
end

tab1.key1 = nil
for k, v in pairs(tab1) do
    print(k .. " - " .. v)
end
```

### 2. boolean (布尔)

boolean 类型只有两个可选值：true (真) 和 false (假)，Lua 把 false 和 nil 看作“假”，

把其他值看作“真”：

```
print(type(true))
print(type(false))
print(type(nil))

if false or nil then
    print("至少有一个是 true")
else
    print("false 和 nil 都为 false!")
end
```

以上代码的执行结果如下：

```
$ lua test.lua
boolean
boolean
nil
false 和 nil 都为 false!
```

### 3. number (数字)

Lua 默认只有一种 number 类型：double（双精度）类型。以下几种写法都被看作 number 类型：

```
print(type(2))
print(type(2.2))
print(type(0.2))
print(type(2e+1))
print(type(0.2e-1))
print(type(7.8263692594256e-06))
```

以上代码的执行结果如下：

```
number
number
number
number
number
number
```

### 4. string (字符串)

字符串由一对双引号或单引号来表示，例如：

```
string1 = "this is string1"
string2 = 'this is string2'
```

也可以用两个方括号“`[[ ]]`”来表示“一块”字符串：

```
html = [[
<html>
<head></head>
<body>
<a href="http://www.w3cschool.cc/">w3cschool 菜鸟教程 </a>
</body>
```

```

</html>
]]
print(html)

```

以上代码的执行结果如下：

```

<html>
<head></head>
<body>
<a href="http://www.w3cschool.cc/">w3cschool 菜鸟教程 </a>
</body>
</html>

```

在对一个数字字符串进行算术操作时，Lua 会尝试将这个数字字符串转成一个数字：

```

>print("2" + 6)
8.0

```

```

>print("2" + "6")
8.0

```

```

>print("2 + 6")
2 + 6

```

```

>print("-2e2" * "6")
-1200.0

```

```

>print("error" + 1)
stdin:1: attempt to perform arithmetic on a string value
stack traceback:
  stdin:1: in main chunk
  [C]: in ?
>

```

以上代码中 “error” + 1 执行报错了，字符串连接使用的是 ..，例如：

```

>print("a" .. "b")
ab

```

```

>print(157 .. 428)
157428
>

```

使用 # 计算字符串的长度，放在字符串前面，如下面实例：

```

>len = "www.google.com"
>print(#len)
14
>print("#www.google.com")
14
>

```

## 5. table (表)

在 Lua 里，table 的创建是通过“构造表达式”完成的，最简单的构造表达式是 {}，用来创建一个空表。也可以在表里添加一些数据，直接初始化表：

```
-- 创建一个空的 table
local tbl1 = {}
```

```
-- 直接初始化表
local tbl2 = {"apple", "pear", "orange", "grape"}
```

Lua 中的表 (table) 其实是一个“关联数组”，数组的索引可以是数字或者字符串：

```
-- table_test.lua 脚本文件
a = {}
a["key"] = "value"
key = 10
a[key] = 22
a[key] = a[key] + 11
for k, v in pairs(a) do
    print(k .. " : " .. v)
end
```

脚本的执行结果如下：

```
$ lua table_test.lua
key : value
10 : 33
```

不同于其他语言的数组把 0 作为数组的初始索引，在 Lua 中，表的默认初始索引一般以 1 开始：

```
-- table_test2.lua 脚本文件
local tbl = {"apple", "pear", "orange", "grape"}
for key, val in pairs(tbl) do
    print("Key", key)
end
```

脚本的执行结果如下：

```
$ lua table_test2.lua
Key 1
Key 2
Key 3
Key 4
```

table 不会固定长度大小，添加新数据时 table 长度会自动增长，没初始化的 table 都是 nil：

```
-- table_test3.lua 脚本文件
a3 = {}
for i = 1, 10 do
    a3[i] = i
end
a3["key"] = "val"
print(a3["key"])
print(a3["none"])
```

脚本的执行结果如下：

```
$ lua table_test3.lua
val
nil
```

## 6. function (函数)

在 Lua 中，函数被看作“第一类值”，函数可以存在变量中：

```
-- function_test.lua 脚本文件
function factorial1(n)
if n == 0 then
    return 1
else
    return n * factorial1(n - 1)
end
end
print(factorial1(5))
factorial2 = factorial1
print(factorial2(5))
```

脚本的执行结果如下：

```
$ lua function_test.lua
120
120
```

function 可以以匿名函数的方式通过参数传递：

```
-- function_test2.lua 脚本文件
function anonymous(tab, fun)
for k, v in pairs(tab) do
    print(fun(k, v))
end
end
tab = { key1 = "val1", key2 = "val2" }
anonymous(tab, function(key, val)
return key .. " = " .. val
end)
```

脚本的执行结果如下：

```
$ lua function_test2.lua
key1 = val1
key2 = val2
```

## 7. thread (线程)

在 Lua 中，最主要的线程是协同程序 (coroutine)。它跟线程 (thread) 差不多，拥有自己独立的栈、局部变量和指令指针，可以跟其他协同程序共享全局变量和其他大部分内容。

线程跟协程的区别：可以同时运行多个线程，而任意时刻只能运行一个协程，并且处于运行状态的协程只有在被挂起 (suspend) 时才会暂停。

## 8. userdata (自定义类型)

userdata 是一种用户自定义数据，用于表示一种由应用程序或 C/C++ 语言库所创建的类



型, 可以将任意 C/C++ 的任意数据类型的数据 (通常是 struct 和指针) 存储到 Lua 变量中调用。

## 6.4 Lua 变量

变量在使用前, 必须在代码中进行声明, 即创建该变量。编译程序执行代码之前, 编译器需要知道如何给语句变量分配存储区, 用于存储变量的值。

Lua 变量有 3 种类型: 全局变量、局部变量、表中的域。

Lua 中的变量是全局变量, 除非用 local 显式声明为局部变量。

局部变量的作用域为从声明位置开始到所在语句块结束。变量的默认值均为 nil。

```
-- test.lua 文件脚本
a = 5                -- 全局变量
local b = 5          -- 局部变量

function joke()
    c = 5             -- 全局变量
    local d = 6       -- 局部变量
end

joke()
print(c,d)           --> 5 nil

do
    local a = 6       -- 局部变量
    b = 6              -- 全局变量
    print(a,b);       --> 6 6
end

print(a,b)           --> 5 6
```

执行以上实例输出结果如下:

```
$ lua test.lua
5      nil
6      6
5      6
```

### 6.4.1 赋值语句

赋值是改变一个变量值和改变表域最基本的方法。

```
a = "hello" .. "world"
t.n = t.n + 1
```

Lua 可以对多个变量同时赋值, 变量列表和值列表的各个元素用逗号分开, 赋值语句右边的值会依次赋给左边的变量。

```
a, b = 10, 2*x      -->   a=10; b=2*x
```

遇到赋值语句, Lua 会先计算右边所有的值, 然后执行赋值操作, 所以可以这样交换变

量的值:

```
x, y = y, x          -- swap 'x' for 'y'
a[i], a[j] = a[j], a[i] -- swap 'a[i]' for 'a[j]'
```

当变量个数和值的个数不一致时, Lua 会一直以变量个数为基础采取以下策略:

- 1) 变量个数 > 值的个数, 按变量个数补足 nil。
- 2) 变量个数 < 值的个数, 多余的值会被忽略。

例如:

```
a, b, c = 0, 1
print(a,b,c)          --> 0  1  nil

a, b = a+1, b+1, b+2   -- b+2 的值被忽略
print(a,b)            --> 1  2

a, b, c = 0
print(a,b,c)          --> 0  nil  nil
```

上面最后一个例子是一个常见的错误情况。注意, 如果要对多个变量赋值, 则必须依次对每个变量赋值。

```
a, b, c = 0, 0, 0
print(a,b,c)          --> 0  0  0
```

多值赋值经常用来交换变量, 或将函数调用返回给变量:

```
a, b = f()
```

f() 返回两个值, 第一个值赋给 a, 第二个值赋给 b。

应该尽可能地使用局部变量, 有两个好处:

- 1) 避免命名冲突。
- 2) 访问局部变量的速度比全局变量更快。

## 6.4.2 索引

对 table 的索引使用方括号 []。Lua 也提供了 . (点) 操作。

```
t[i]
t.i          -- 当索引为字符串类型时的一种简化写法
gettable_event(t,i) -- 采用索引访问本质上是一个类似这样的函数调用
```

例如:

```
>site = {}
>site["key"] = "www.w3cschool.cc"
>print(site["key"])
www.w3cschool.cc
>print(site.key)
www.w3cschool.cc
```

## 6.5 Lua 循环

很多情况下需要做一些有规律性的重复操作,因此在程序中就需要重复执行某些语句。一组被重复执行的语句称为循环体,能否继续重复,由条件决定。循环结构是在一定条件下反复执行某段程序的流程结构,被反复执行的程序称为循环体。循环语句是由循环体及循环的终止条件两部分组成的。循环语句的结构如图 6-1 所示。

Lua 语言提供了几种循环语句,如表 6-3 所示。

表 6-3 Lua 循环语句

类型	描述
while 循环	在条件为 true 时,程序重复地执行某些语句。执行语句前先检查条件是否为 true
for 循环	重复执行指定语句,重复次数可以在 for 语句中指定
repeat...until	重复执行循环,直到指定的条件为真时为止
循环嵌套	可以在循环内嵌套一个或多个循环语句

### 1. 循环控制语句

循环控制语句用于控制程序的流程,以实现程序的各种结构方式。

Lua 支持以下循环控制语句:

- break 语句:退出当前循环或语句,开始执行紧接着的语句。

### 2. 无限循环

在循环体中,如果条件永远为 true 循环语句就会永远执行下去,以下以 while 循环为例:

```
while( true )
do
    print(" 循环将永远执行下去 ")
end
```

## 6.6 Lua 流程控制

Lua 编程语言流程控制语句通过程序设定一个或多个条件语句来设定。在条件为 true 时执行指定程序代码,在条件为 false 时执行其他指定代码。

图 6-2 是循环语句流程控制流程图。

控制结构的条件表达式结果可以是任何值,Lua 认为 false 和 nil 为假,true 和非 nil 为真。

要注意的是,在 Lua 中 0 为 true:

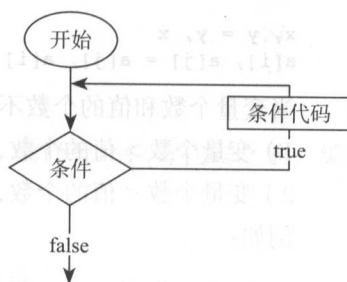


图 6-1 循环语句的结构

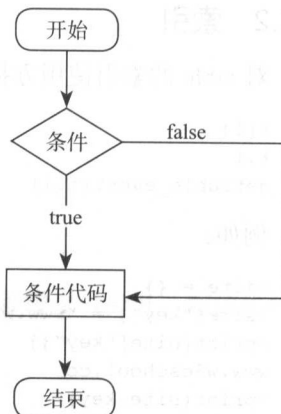


图 6-2 循环语句流程控制流程图

```
--[ 0 为 true ]
if(0)
then
    print("0 为 true")
end
```

以上代码的输出结果如下：

```
0 为 true
```

Lua 提供了以下控制结构语句。

- if 语句：由一个表达式作为条件判断，紧跟其他语句。
- if...else 语句：可以与 else 搭配使用，在 if 条件表达式为 false 时，执行 else 语句代码。
- if 嵌套语句：可以在 if 或 else if 中使用一个或多个 if 或 else if 语句。

## 6.7 Lua 函数

在 Lua 中，函数是对语句和表达式进行抽象的主要方法，既可以用来处理一些特殊的工作，也可以用来计算一些值。

Lua 提供了许多内建函数，可以很方便地在程序中调用它们，如 print() 函数可以将传入的参数输出到控制台上。

Lua 函数主要有两种用途。

- 1) 完成指定的任务：这种情况下函数作为调用语句使用。
- 2) 计算并返回值：这种情况下函数作为赋值语句的表达式使用。

### 6.7.1 函数的定义

Lua 编程语言函数定义格式如下：

```
optional_function_scope function function_name( argument1, argument2,
argument3,..., argumentn)
    function_body
    return result_params_comma_separated
end
```

说明：

- optional\_function\_scope：该参数是可选的，指定函数是全局函数还是局部函数。未设置该参数默认为全局函数；如果需要设置函数为局部函数，则需要使用关键字 local。
- function\_name：指定函数名称。
- argument1, argument2, argument3, ..., argumentn：函数参数，多个参数以逗号隔开，函数也可以不带参数。

- `function_body`: 函数体, 即函数中需要执行的代码语句块。
- `result_params_comma_separated`: 函数返回值, Lua 语言函数可以返回多个值, 每个值以逗号隔开。

下面是一个函数实例, 实例定义了函数 `max()`, 参数为 `num1`、`num2`, 用于比较两个值的大小, 并返回最大值。

```
--[[ 函数返回两个值的最大值 --]]
function max(num1, num2)

    if (num1 > num2) then
        result = num1;
    else
        result = num2;
    end

    return result;
end
-- 调用函数
print(" 两值比较最大值为 ",max(10,4))
print(" 两值比较最大值为 ",max(5,6))
```

以上代码的执行结果如下:

```
两值比较最大值为      10
两值比较最大值为      6
```

Lua 中可以将函数作为参数传递给函数, 例如:

```
myprint = function(param)
    print(" 这是打印函数 -   ##",param,"##")
end

function add(num1,num2,functionPrint)
    result = num1 + num2
    -- 调用传递的函数参数
    functionPrint(result)
end
myprint(10)
-- myprint 函数作为参数传递
add(2,5,myprint)
```

以上代码的执行结果如下:

```
这是打印函数 -   ##      10      ##
这是打印函数 -   ##      7      ##
```

## 6.7.2 多返回值

Lua 函数可以返回多个结果值, 如 `string.find`, 其返回匹配串开始和结束的下标 (如果不存在匹配串则返回 `nil`)。

```
>s, e = string.find("www.google.com", "google")
>print(s, e)
5      10
```

Lua 函数中，在 return 后，返回值可以是返回多值，例如：

```
function maximum (a)
    local mi = 1          -- 最大值索引
    local m = a[mi]       -- 最大值
    for i, val in ipairs(a) do
        if val > m then
            mi = i
            m = val
        end
    end
    return m, mi
end
```

```
print(maximum({8,10,23,12,5}))
```

以上代码的执行结果如下：

```
23      3
```

### 6.7.3 可变参数

Lua 函数可以接受可变数目的参数，和 C 语言类似，在函数参数列表中使用“...”表示函数有可变的参数。

Lua 将函数的参数放在 arg 表中，#arg 表示传入参数的个数。

例如，计算几个数的平均值可以这样操作：

```
function average(...)
    result = 0
    local arg={...}
    for i,v in ipairs(arg) do
        result = result + v
    end
    print(" 总共传入 " .. #arg .. " 个数 ")
    return result/#arg
end
```

```
print(" 平均值为 ", average(10,5,3,4,5,6))
```

以上代码的执行结果如下：

```
总共传入 6 个数
平均值为      5.5
```

## 6.8 Lua 运算符

运算符是一个特殊的符号，用于告诉解释器执行特定的数学或逻辑运算。Lua 提供了以

下几种运算符类型：算术运算符、关系运算符、逻辑运算符、其他运算符。

6.8.1    算术运算符

表 6-4 列出了 Lua 语言中的常用算术运算符，设定 A 的值为 10，B 的值为 20。  
可以通过以下实例来更加透彻地理解算术运算符的应用：

```
a = 21
b = 10
c = a + b
print("Line 1 - c 的值为 ", c )
c = a - b
print("Line 2 - c 的值为 ", c )
c = a * b
print("Line 3 - c 的值为 ", c )
c = a / b
print("Line 4 - c 的值为 ", c )
c = a % b
print("Line 5 - c 的值为 ", c )
c = a^2
print("Line 6 - c 的值为 ", c )
c = -a
print("Line 7 - c 的值为 ", c )
```

表 6-4    常用算术运算符

运算符	描述	实例
+	加法	A+B 输出结果为 30
-	减法	A-B 输出结果为 -10
*	乘法	A*B 输出结果为 200
/	除法	B/A 输出结果为 2
%	取余	B%A 输出结果为 0
^	幂	A^2 输出结果为 100
-	负号	-A 输出结果为 -10

以上程序的执行结果如下：

```
Line 1 - c 的值为    31
Line 2 - c 的值为    11
Line 3 - c 的值为    210
Line 4 - c 的值为    2.1
Line 5 - c 的值为    1
Line 6 - c 的值为    441
Line 7 - c 的值为    -21
```

6.8.2    关系运算符

表 6-5 列出了 Lua 语言中的常用关系运算符，设定 A 的值为 10，B 的值为 20。

表 6-5    常用关系运算符

运算符	描述	实例
==	等于。如果两个值相等则返回 true，否则返回 false	A==B 输出结果为 false
~=	不等于。如果两个值相等则返回 false，否则返回 true	A~=B 输出结果为 true
>	大于。如果左边值大于右边值则返回 true，否则返回 false	A>B 输出结果为 false
<	小于。如果左边值小于右边值则返回 false，否则返回 true	A<B 输出结果为 true
>=	大于等于。如果左边值大于等于右边值则返回 true，否则返回 false	A>=B 输出结果为 false
<=	小于等于。如果左边值小于等于右边值则返回 true，否则返回 false	A<=B 输出结果为 true

可以通过以下实例来更加透彻地理解关系运算符的应用：

```
a = 21
```



```

b = 10

if ( a == b )
then
    print("Line 1 - a 等于 b" )
else
    print("Line 1 - a 不等于 b" )
end

if ( a ~= b )
then
    print("Line 2 - a 不等于 b" )
else
    print("Line 2 - a 等于 b" )
end

if ( a < b )
then
    print("Line 3 - a 小于 b" )
else
    print("Line 3 - a 大于等于 b" )
end

if ( a > b )
then
    print("Line 4 - a 大于 b" )
else
    print("Line 5 - a 小于等于 b" )
end

-- 修改 a 和 b 的值
a = 5
b = 20
if ( a <= b )
then
    print("Line 5 - a 小于等于 b" )
end

if ( b >= a )
then
    print("Line 6 - b 大于等于 a" )
end

```

以上程序的执行结果如下：

```

Line 1 - a 不等于 b
Line 2 - a 不等于 b
Line 3 - a 大于等于 b
Line 4 - a 大于 b
Line 5 - a 小于等于 b
Line 6 - b 大于等于 a

```

### 6.8.3 逻辑运算符

表 6-6 列出了 Lua 语言中的常用逻辑运算符，设定 A 的值为 true，B 的值为 false。

表 6-6 常用逻辑运算符

运算符	描述	实例
and	逻辑与操作。如果两个值都为 true，则返回 true	A and B 输出结果为 false
or	逻辑或操作。如果两个值中的任意一个值为 true，则返回 true	A or B 输出结果为 true
not	逻辑非操作。取反操作，如果条件为 true，则返回 false；如果条件为 false，则返回 true	not (A and B) 输出结果为 true

可以通过以下实例来更加透彻地理解逻辑运算符的应用：

```

a = true
b = true

if ( a and b )
then
    print("a and b - 条件为 true" )
end

if ( a or b )
then
    print("a or b - 条件为 true" )
end

print("----- 分割线 -----" )

-- 修改 a 和 b 的值
a = false
b = true

if ( a and b )
then
    print("a and b - 条件为 true" )
else
    print("a and b - 条件为 false" )
end

if ( not( a and b ) )
then
    print("not( a and b ) - 条件为 true" )
else
    print("not( a and b ) - 条件为 false" )
end

```

以上程序的执行结果如下：

```

a and b - 条件为 true
a or b - 条件为 true
----- 分割线 -----
a and b - 条件为 false
not( a and b ) - 条件为 true

```

#### 6.8.4 其他运算符

表 6-7 列出了 Lua 语言中的连接运算符与计算表或字符串长度的运算符。

表 6-7 其他运算符

运算符	描述	实例
..	连接两个字符串	a..b, 其中 a 为 “Hello”, b 为 “World”, 输出结果为 “Hello World”
#	一元运算符, 返回字符串或表的长度	# “Hello” 返回 5

可以通过以下实例来更加透彻地理解连接运算符与计算表或字符串长度的运算符的应用:

```
a = "Hello "
b = "World"

print(" 连接字符串 a 和 b ", a..b )
print("b 字符串长度 ", #b )
print(" 字符串 Test 长度 ", #"Test" )
print("google 网址长度 ", #"www.google.com" )
```

以上程序的执行结果如下:

```
连接字符串 a 和 b      Hello World
b 字符串长度          5
字符串 Test 长度      4
google 网址长度       14
```

### 6.8.5 运算符的优先级

运算符的优先级从高到低的顺序如下:

```
^
not    - (unary)
*      /
+      -
..
<><=   >=   ~=   ==
and
or
```

除 ^ 和 .. 以外的二元运算符都是左连接的。

示例:

```
a+i < b/2+1      <-->      (a+i) < ((b/2)+1)
5+x^2*8          <-->      5+((x^2)*8)
a < y and y <= z  <-->      (a < y) and (y <= z)
-x^2              <-->      -(x^2)
x^y^z             <-->      x^(y^z)
```

可以通过以下实例来更加透彻地了解 Lua 语言运算符的优先级:

```
a = 20
b = 10
c = 15
d = 5
```

```

e = (a + b) * c / d; -- (30 * 15) / 5
print("(a + b) * c / d 运算值为 :", e)

e = ((a + b) * c) / d; -- (30 * 15) / 5
print("((a + b) * c) / d 运算值为 :", e)

e = (a + b) * (c / d); -- (30) * (15/5)
print("(a + b) * (c / d) 运算值为 :", e)

e = a + (b * c) / d; -- 20 + (150/5)
print("a + (b * c) / d 运算值为 :", e)

```

以上程序的执行结果如下:

```

(a + b) * c / d 运算值为      :    90.0
((a + b) * c) / d 运算值为    :    90.0
(a + b) * (c / d) 运算值为    :    90.0
a + (b * c) / d 运算值为      :    50.0

```

## 6.9 Lua 字符串

字符串或串 (string) 是由数字、字母、下划线组成的一串字符。

Lua 语言中字符串可以使用以下 3 种方式来表示:

- 单引号间的一串字符;
- 双引号间的一串字符;
- [[ 和 ]] 间的一串字符。

以上 3 种方式的字符串实例如下:

```

string1 = "Lua"
print("\" 字符串 1 是 \"", string1)
string2 = 'google.com'
print("字符串 2 是 ", string2)

```

```

string3 = [[ "天天向上" ]]
print("字符串 3 是 ", string3)

```

以上代码执行结果如下:

```

" 字符串 1 是 " Lua
字符串 2 是   google.com
字符串 3 是   "天天向上"

```

转义字符用于表示不能直接显示的字符, 如后退键、回车键等。例如, 在字符串中转换双引号可以使用 “\”。

表 6-8 描述了所有的转义字符及其所对应的意义。

表 6-8 Lua 转义字符

转义字符	意义	ASCII 码 (十进制)
\a	响铃 (BEL)	007

(续)

转义字符	意义	ASCII 码 (十进制)
\b	退格 (BS), 将当前位置移到前一系列	008
\f	换页 (FF), 将当前位置移到下页开头	012
\n	换行 (LF), 将当前位置移到下一行开头	010
\r	回车 (CR), 将当前位置移到本行开头	013
\t	水平制表 (HT)(跳到下一个 Tab 位置)	009
\v	垂直制表 (VT)	011
\\	代表一个反斜线字符 “\”	092
\'	代表一个单引号 (撇号) 字符	039
\"	代表一个双引号字符	034
\0	空字符 (NULL)	000
\ddd	1 到 3 位八进制数所代表的任意字符	3 位八进制
\xhh	1 到 2 位十六进制数所代表的任意字符	2 位十六进制

## 6.10 Lua 数组

数组, 就是相同数据类型的元素按一定顺序排列的集合, 包括一维数组和 multidimensional 数组。Lua 数组的索引值可以使用整数表示, 数组的大小不是固定的。

### 6.10.1 一维数组

一维数组是最简单的数组, 其逻辑结构是线性表。一维数组可以用 for 循环输出数组中的元素, 如下实例:

```
array = {"Lua", "Tutorial"}

for i= 0, 2 do
    print(array[i])
end
```

以上代码的执行结果如下:

```
nil
Lua
Tutorial
```

可以使用整数索引来访问数组元素。如果索引没有值, 则返回 nil。

Lua 索引值以 1 为起始值, 也可以指定从 0 开始。另外, 还可以用负数作为数组索引值, 例如:

```
array = {}

for i= -2, 2 do
    array[i] = i * 2
```

```
end
```

```
for i = -2,2 do
    print(array[i])
end
```

以上代码的执行结果如下:

```
-4
-2
0
2
4
```

### 6.10.2 多维数组

多维数组即数组中包含数组或一维数组的索引对应一个数组。

以下是一个三行三列的阵列多维数组:

```
-- 初始化数组
array = {}
for i=1,3 do
    array[i] = {}
    for j=1,3 do
        array[i][j] = i*j
    end
end

-- 访问数组
for i=1,3 do
    for j=1,3 do
        print(array[i][j])
    end
end
```

以上代码的执行结果如下:

```
1
2
3
2
4
6
3
6
9
```

不同索引的三行三列阵列多维数组:

```
-- 初始化数组
array = {}
maxRows = 3
maxColumns = 3
for row=1,maxRows do
```

```

for col=1,maxColumns do
    array[row*maxColumns +col] = row*col
end
end

```

```

-- 访问数组
for row=1,maxRows do
    for col=1,maxColumns do
        print(array[row*maxColumns +col])
    end
end
end

```

以上代码的执行结果如下：

```

1
2
3
2
4
6
3
6
9

```

上面的实例中，数组设定了指定的索引值，这样可以避免出现 nil 值，有利于节省内存空间。

## 6.11 Lua 迭代器

迭代器 (iterator) 是一种对象，用来遍历标准模板库容器中的部分或全部元素。每个迭代器对象代表容器中确定的地址。

在 Lua 中，迭代器是一种支持指针类型的结构，它可以遍历集合的每一个元素。

### 6.11.1 泛型 for 迭代器

泛型 for 迭代器在自己内部保存迭代函数，实际上它保存 3 个值：迭代函数、状态常量、控制变量。

泛型 for 迭代器提供了集合的 key/value 对，语法格式如下：

```

for k, v in pairs(t) do
    print(k, v)
end

```

上面代码中，k,v 为变量列表，pair(t) 为表达式列表。

查看以下实例：

```

array = {"Lua", "Tutorial"}

for key,value in ipairs(array) do

```



```

    print(key, value)
end

```

以上代码的执行结果如下：

```

1  Lua
2  Tutorial

```

以上实例中，使用了 Lua 默认提供的迭代函数 `ipairs`。

泛型 `for` 迭代器的执行过程：

1) 初始化，计算 `in` 后面表达式的值，表达式应该返回泛型 `for` 迭代器需要的 3 个值：迭代函数、状态常量、控制变量。与多值赋值一样，如果表达式返回的结果个数不足 3 个，则自动用 `nil` 补足，多出部分会被忽略。

2) 将状态常量和控制变量作为参数调用迭代函数（注意：对于 `for` 结构来说，状态常量没有用处，仅仅在初始化时获取它的值并传递给迭代函数）。

3) 将迭代函数返回的值赋给变量列表。

4) 如果返回的第一个值为 `nil`，则循环结束，否则执行循环体。

5) 回到第二步再次调用迭代函数。

在 Lua 中常常使用函数来描述迭代器，每次调用该函数就返回集合的下一个元素。Lua 的迭代器包括两种类型：无状态的迭代器和多状态的迭代器。

### 6.11.2 无状态的迭代器

无状态的迭代器是指不保留任何状态的迭代器。在循环中可以利用无状态迭代器避免创建闭包花费额外的代价。

每一次迭代，迭代函数均用两个变量（状态常量和控制变量）的值作为参数被调用，一个无状态的迭代器利用这两个值可以获取下一个元素。

无状态的迭代器的典型的例子是 `ipairs`，可遍历数组的每一个元素。

以下实例使用了一个简单的函数来实现迭代器，实现数字 `n` 的平方：

```

function square(iteratorMaxCount,currentNumber)
if currentNumber<iteratorMaxCount
then
    currentNumber = currentNumber+1
return currentNumber, currentNumber*currentNumber
end
end

for i,n in square,3,0
do
    print(i,n)
end

```

以上代码的执行结果如下：

```

1    1

```

```

2      4
3      9

```

迭代的状态包括被遍历的表（循环过程中不会改变的状态常量）和当前的索引下标（控制变量），`ipairs` 和迭代函数都很简单，在 Lua 中可以这样实现：

```

function iter (a, i)
    i = i + 1
    local v = a[i]
    if v then
        return i, v
    end
end

function ipairs (a)
    return iter, a, 0
end

```

当 Lua 调用 `ipairs(a)` 开始循环时，可获取 3 个值：迭代函数 `iter`、状态常量 `a`、控制变量初始值 0。然后 Lua 调用 `iter(a,0)` 返回 1、`a[1]`（除非 `a[1]=nil`）。第二次迭代调用 `iter(a,1)` 返回 2、`a[2]`……直到第一个 `nil` 元素。

### 6.11.3 多状态的迭代器

很多情况下，迭代器需要保存多个状态信息而不是简单的状态常量和控制变量，最简单的方法是使用闭包，还有一种方法是将所有的状态信息封装到表内，将表作为迭代器的状态常量，因为这种情况下可以将所有的信息存放在表内，所以迭代函数通常不需要第二个参数。

以下实例创建了自己的迭代器：

```

array = {"Lua", "Tutorial"}

function elementIterator (collection)
    local index = 0
    local count = #collection
    -- 闭包函数
    return function ()
        index = index + 1
        if index <= count
        then
            -- 返回迭代器的当前元素
            return collection[index]
        end
    end
end

for element in elementIterator(array)
do
    print(element)
end

```

以上代码的执行结果如下：

```
Lua
Tutorial
```

以上实例中可以看到，`elementIterator` 内使用了闭包函数，实现计算集合大小并输出各个元素。

## 6.12 Lua 表

表（table）是 Lua 的一种数据结构，用来创建不同的数据类型，如数字、字典等。Lua 表使用关联型数组，可以使用任意类型的值作为数组的索引，但这个值不能是 `nil`。Lua 表不是固定大小的，可以根据自己的需要进行扩容。Lua 本身也是通过表解决模块、包和对象的。例如，`string.format` 表示使用“format”索引表 `string`。

构造器是创建和初始化表的表达式。最简单的构造函数是 `{}`，用来创建一个空表。可以直接初始化数组：

```
-- 初始化表
mytable = {}

-- 指定值
mytable[1] = "Lua"

-- 移除引用
mytable = nil
-- Lua 垃圾回收会释放内存
```

首先为表 `a` 设置元素，然后将 `a` 赋值给 `b`，则 `a` 与 `b` 都指向同一个内存。如果 `a` 设置为 `nil`，则 `b` 同样能访问 `table` 的元素。如果没有指定的变量指向 `a`，Lua 的垃圾回收机制会清理相对应的内存。

以下实例演示了上面描述的情况：

```
-- 简单的 table
mytable = {}
print("mytable 的类型是 ", type(mytable))

mytable[1] = "Lua"
mytable["wow"] = "修改前"
print("mytable 索引为 1 的元素是 ", mytable[1])
print("mytable 索引为 wow 的元素是 ", mytable["wow"])

-- alternatetable 和 mytable 的是指同一个 table
alternatetable = mytable

print("alternatetable 索引为 1 的元素是 ", alternatetable[1])
print("mytable 索引为 wow 的元素是 ", alternatetable["wow"])

alternatetable["wow"] = "修改后"
```

```
print("mytable 索引为 wow 的元素是 ", mytable["wow"])
```

```
-- 释放变量
```

```
alternatetable = nil
```

```
print("alternatetable 是 ", alternatetable)
```

```
-- mytable 仍然可以访问
```

```
print("mytable 索引为 wow 的元素是 ", mytable["wow"])
```

```
mytable = nil
```

```
print("mytable 是 ", mytable)
```

以上代码的执行结果如下：

```
mytable 的类型是      table
mytable 索引为 1 的元素是      Lua
mytable 索引为 wow 的元素是      修改前
alternatetable 索引为 1 的元素是      Lua
mytable 索引为 wow 的元素是      修改前
mytable 索引为 wow 的元素是      修改后
alternatetable 是      nil
mytable 索引为 wow 的元素是      修改后
mytable 是      nil
```

## 6.13 Lua 模块与包

模块类似于一个封装库，从 Lua 5.1 开始，Lua 加入了标准的模块管理机制，可以把一些公用的代码放在一个文件里，以 API 接口的形式在其他地方调用，有利于代码的重用和降低代码耦合度。

Lua 的模块是由变量、函数等已知元素组成的表，因此创建一个模块很简单，就是创建一个表，然后把需要导出的常量、函数放入其中，最后返回这个表。

以下实例为创建自定义模块 module.lua，文件代码格式如下：

```
-- 文件名为 module.lua
```

```
-- 定义一个名为 module 的模块
```

```
module = {}
```

```
-- 定义一个常量
```

```
module.constant = "这是一个常量"
```

```
-- 定义一个函数
```

```
function module.func1()
```

```
    io.write("这是一个公有函数! \n")
```

```
end
```

```
local function func2()
```

```
    print("这是一个私有函数! ")
```

```
end
```

```
function module.func3()
```

```
func2()
end
```

```
return module
```

由上面实例可知，模块的结构就是一个表结构，因此可以像操作表里的元素一样调用模块里的常量和函数。

上例的 func2 声明为程序块的局部变量，表示一个私有函数，因此不能从外部访问模块的这个私有函数，必须通过模块公有函数调用。

### 6.13.1 require 函数

Lua 提供 require 函数用来加载模块。要加载一个模块，只需要简单地调用 require 函数即可。例如：

```
require("<模块名>")
```

或者

```
require "<模块名>"
```

执行 require 函数后会返回一个由模块常量或函数组成的表，并且还会定义一个包含该表的全局变量。

```
-- test_module.lua 文件
-- module 模块为上文提到到 module.lua
require("module")

print(module.constant)

module.func3()
```

以上代码的执行结果如下：

```
这是一个常量
这是一个私有函数!
```

还可以给加载的模块定义一个别名变量，方便调用：

```
-- test_module2.lua 文件
-- module 模块为上文提到到 module.lua
-- 别名变量 m
local m = require("module")

print(m.constant)

m.func3()
```

以上代码的执行结果如下：

```
这是一个常量
这是一个私有函数!
```

在 Nginx 中使用 Lua，为了避免全局变量被不同的例程继承并改变，从而出现变量被修改或删除的错误，推荐以别名形式载入模块，即使用 `local` 为库定义一个局部变量。

### 6.13.2 加载机制

对于自定义的模块，模块文件不是放在哪个文件目录都行，`require` 函数有自己的文件路径加载策略，它会尝试从 Lua 文件或 C 程序库中加载模块。

`require` 用于搜索 Lua 文件的路径存放在 `package.path` 这个全局变量中。当 Lua 启动后，会以环境变量 `LUA_PATH` 的值初始这个环境变量。如果没有找到该环境变量，则使用编译时定义的默认路径初始化。

当然，如果没有 `LUA_PATH` 这个环境变量，也可以自定义设置，在当前用户根目录下打开 `.profile` 文件（没有则创建，打开 `.bashrc` 文件也可以）。例如，把 “`~/lua/`” 路径加入 `LUA_PATH` 环境变量中：

```
#LUA_PATH
export LUA_PATH="~/lua/?..lua;;"
```

文件路径以 “`;`” 分隔，最后的两个 “`;;`” 表示新加的路径后面加上原来的默认路径。接着，更新环境变量参数，使之立即生效。

```
source ~/.profile
```

这时假设 `package.path` 的值是

```
/Users/dengjoe/lua/?..lua;./?.lua;/usr/local/share/lua/5.1/?..lua;/usr/local/share/lua/5.1/?/init.lua;/usr/local/lib/lua/5.1/?..lua;/usr/local/lib/lua/5.1/?/init.lua
```

那么调用 `require(“module”)` 时就会尝试打开以下文件目录去搜索目标。

```
/Users/dengjoe/lua/module.lua;
./module.lua
/usr/local/share/lua/5.1/module.lua
/usr/local/share/lua/5.1/module/init.lua
/usr/local/lib/lua/5.1/module.lua
/usr/local/lib/lua/5.1/module/init.lua
```

如果找到目标文件，则会调用 `package.loadfile` 加载模块；否则，就会去找 C 程序库。

搜索的文件路径从全局变量 `package.cpath` 获取，而这个变量则通过环境变量 `LUA_CPATH` 初始化。

搜索策略跟上面一样，只不过搜索的是 `so` 或 `dll` 类型的文件。如果找到所需文件，那么 `require` 就会通过 `package.loadlib` 来加载它。

### 6.13.3 C 包

Lua 和 C 是很容易结合的，可以使用 C 为 Lua 写包。与在 Lua 中写包不同，C 包在使用以前必须首先加载并连接，在大多数系统中最容易的实现方式是通过动态连接库机制加

载并连接。

Lua 在 `loadlib` 函数内提供了所有的动态连接功能。这个函数有两个参数：库的绝对路径和初始化函数。典型的调用实例如下：

```
local path = "/usr/local/lua/lib/libluasocket.so"
local f = loadlib(path, "luaopen_socket")
```

`loadlib` 函数加载指定的库并且连接到 Lua，然而它并不打开库（也就是说没有调用初始化函数），反之它返回初始化函数作为 Lua 的一个函数，这样可以直接在 Lua 中调用它。

如果加载动态库或者查找初始化函数时出错，`loadlib` 将返回 `nil` 和错误信息。可以通过修改前面一段代码，使其检测错误然后调用初始化函数：

```
local path = "/usr/local/lua/lib/libluasocket.so"
-- 或者 path = "C:\\windows\\luasocket.dll", 这是 Window 平台下
local f = assert(loadlib(path, "luaopen_socket"))
f() -- 真正打开库
```

一般情况下，期望二进制的发布库包含一个与前面代码段相似的 `stub` 文件，安装二进制库的时候可以随便放在某个目录，只需要修改 `stub` 文件对应二进制库的实际路径即可。

将 `stub` 文件所在的目录加入到 `LUA_PATH`，这样设定后就可以使用 `require` 函数加载 C 库了。

## 6.14 Lua 元表

在 Lua 表中可以通过访问 `key` 得到对应的 `value` 值，但是无法对两个表进行操作。因此 Lua 提供了元表（`metatable`），允许改变表的行为，每个行为关联了对应的元方法。

例如，使用元表可以定义 Lua 如何计算两个表的相加操作 `a+b`。

当 Lua 试图对两个表进行相加时，先检查两者之一是否有元表，之后检查是否有一个包含 “`_add`” 字段，若找到，则调用对应的值。“`_add`” 等即是字段对应的值（往往是一个函数或是表）就是“元方法”。

有两个很重要的函数辅助元表处理：

- `setmetatable(table,metatable)`：对指定表设置元表，如果元表中存在 `_metatable` 键值，`setmetatable` 会失败。
- `getmetatable(table)`：返回对象的元表。

以下实例演示了如何对指定的表设置元表：

```
mytable = {}
mymetatable = {}
setmetatable(mytable,mymetatable)
-- 普通表
-- 元表
-- 把 mymetatable 设为 mytable 的元表
```

以上代码也可以直接写成一行：

```
mytable = setmetatable({}, {})
```



以下为返回对象元表：

```
getmetatable(mytable) -- 返回 mymetatable
```

### 6.14.1 `_index` 元方法

`_index` 键是元表最常用的键。当通过键访问表的时候，如果这个键没有值，那么 Lua 就会寻找该表的元表（假定有 `metatable`）中的 `_index` 键。如果 `_index` 包含一个表格，Lua 会在表格中查找相应的键。

可以使用 Lua 命令进入交互模式查看：

```
$ lua
Lua 5.3.0 Copyright (C) 1994-2015 Lua.org, PUC-Rio
>other = { foo = 3 }
> t = setmetatable({}, { _index = other })
> t.foo
3
> t.bar
nil
```

如果 `_index` 包含一个函数，则 Lua 会调用这个函数，`table` 和键作为参数传递给函数。

`_index` 元方法查看表中元素是否存在：如果不存在，则返回结果为 `nil`；如果存在，则由 `_index` 返回结果。

```
mytable = setmetatable({key1 = "value1"}, {
  _index = function(mytable, key)
    if key == "key2" then
      return "metatablevalue"
    else
      return nil
    end
  end
})
```

```
print(mytable.key1,mytable.key2)
```

以上代码的执行结果如下：

```
value1      metatablevalue
```

实例解析：

- `mytable` 表赋值为 `{key1= "value1" }`。
- `mytable` 设置了元表，元方法为 `_index`。
- 在 `mytable` 表中查找 `key1`，如果找到，则返回该元素，否则继续。
- 在 `mytable` 表中查找 `key2`，如果找到，则返回 `metatablevalue`，否则继续。
- 判断元表有没有 `_index` 方法，如果 `_index` 方法是一个函数，则调用该函数。
- 在元方法中查看是否传入“`key2`”参数（`mytable.key2` 已设置），如果传入“`key2`”参数则返回“`metatablevalue`”，否则返回 `mytable` 对应键值。

可以将以上代码简单写成：

```
mytable = setmetatable({key1 = "value1"}, { _index = { key2 = "metatablevalue" } })
print(mytable.key1, mytable.key2)
```

Lua 查找一个表元素时的规则遵循如下 3 个步骤：

- 1) 在表中查找，如果找到，则返回该元素，否则继续。
- 2) 判断该表是否有元表，如果没有元表，则返回 nil，否则继续。
- 3) 判断元表有没有 \_index 方法。如果 \_index 方法为 nil，则返回 nil；如果 \_index 方法是一个表，则重复第 1 ~ 3 步；如果 \_index 方法是一个函数，则返回该函数的返回值。

### 6.14.2 \_newindex 元方法

\_newindex 元方法用来对表进行更新，\_index 用来对表访问。当给表的一个不存在的索引赋值时，解释器会查找 \_newindex 元方法，如果 \_newindex 元方法存在则调用这个函数而不进行赋值操作。

以下实例演示了 \_newindex 元方法的应用：

```
mymetatable = {}
mytable = setmetatable({key1 = "value1"}, { _newindex = mymetatable })

print(mytable.key1)

mytable.newkey = "新值 2"
print(mytable.newkey, mymetatable.newkey)

mytable.key1 = "新值 1"
print(mytable.key1, mymetatable.key1)
```

以上代码的执行结果如下：

```
value1
nil      新值 2
新值 1    nil
```

以上实例中表设置了元方法 \_newindex，在对新索引键 (newkey) 赋值时 (mytable.newkey = "新值 2")，会调用元方法，而不进行赋值。而如果索引键 (key1) 已存在，则会进行赋值而不调用元方法 \_newindex。

以下实例使用了 rawset 函数更新表：

```
mytable = setmetatable({key1 = "value1"}, {
    _newindex = function(mytable, key, value)
        rawset(mytable, key, "\"" .. value .. "\"")
    end
})

mytable.key1 = "new value"
mytable.key2 = 4
```

```
print(mytable.key1,mytable.key2)
```

以上代码的执行结果如下：

```
new value      "4"
```

6.14.3 为表添加运算符

以下实例演示了两表相加操作：

```
-- 计算表中最大值，table.maxn 在 Lua 5.2 以上版本中已无法使用
-- 自定义计算表中最大值函数 table_maxn
function table_maxn(t)
  local mn = 0
  for k, v in pairs(t) do
    if mn < k then
      mn = k
    end
  end
  return mn
end

-- 两表相加操作
mytable = setmetatable({ 1, 2, 3 }, {
  _add = function(mytable, newtable)
    for i = 1, table_maxn(newtable) do
      table.insert(mytable, table_maxn(mytable)+1,newtable[i])
    end
    return mytable
  end
})
```

```
secondtable = {4,5,6}

mytable = mytable + secondtable
for k,v in ipairs(mytable) do
  print(k,v)
end
```

以上代码的执行结果如下：

```
1      1
2      2
3      3
4      4
5      5
6      6
```

\_add 键包含在元表中，并进行相加操作。元表的提供运算类操作如表 6-9 所示。

表 6-9 元表提供的运算类操作

运算类操作	描述
_ad	对应运算符 “+”
_sub	对应运算符 “-”
_mul	对应运算符 “*”
_div	对应运算符 “/”
_mod	对应运算符 “%”
_unm	对应运算符 “-”
_concat	对应运算符 “..”
_eq	对应运算符 “==”
_lt	对应运算符 “<”
_le	对应运算符 “<=”

6.14.4 \_call 元方法

\_call 元方法在 Lua 调用一个值时调用。以下实例演示了计算表中元素的和：

-- 计算表中最大值, `table.maxn` 在 Lua 5.2 以上版本中已无法使用

-- 自定义计算表中最大值函数 `table_maxn`

```
function table_maxn(t)
```

```
    local mn = 0
```

```
    for k, v in pairs(t) do
```

```
        if mn < k then
```

```
            mn = k
```

```
        end
```

```
    end
```

```
    return mn
```

```
end
```

-- 定义元方法 `_call`

```
mytable = setmetatable({10}, {
```

```
    _call = function(mytable, newtable)
```

```
        sum = 0
```

```
        for i = 1, table_maxn(mytable) do
```

```
            sum = sum + mytable[i]
```

```
        end
```

```
        for i = 1, table_maxn(newtable) do
```

```
            sum = sum + newtable[i]
```

```
        end
```

```
        return sum
```

```
    end
```

```
})
```

```
newtable = {10,20,30}
```

```
print(mytable(newtable))
```

以上代码的执行结果如下:

```
70
```

### 6.14.5 `_tostring` 元方法

`_tostring` 元方法用于修改表的输出行为。以下实例自定义了表的输出内容:

```
mytable = setmetatable({ 10, 20, 30 }, {
```

```
    _tostring = function(mytable)
```

```
        sum = 0
```

```
        for k, v in pairs(mytable) do
```

```
            sum = sum + v
```

```
        end
```

```
        return " 表所有元素的和为 " .. sum
```

```
    end
```

```
})
```

```
print(mytable)
```

以上代码的执行结果如下:

```
表所有元素的和为 60
```

从以上内容可以知道元表可以很好地简化代码, 所以了解 Lua 元表, 可以写出更加简

单优秀的 Lua 代码。

## 6.15 Lua 协同程序

Lua 协同程序与线程类似：拥有独立的堆栈、独立的局部变量、独立的指令指针，同时又与其他协同程序共享全局变量和其他大部分内容。

线程与协同程序的主要区别在于：一个具有多个线程的程序可以同时运行几个线程，而协同程序却需要彼此协作运行。在任一指定时刻只有一个协同程序在运行，并且这个正在运行的协同程序只有在明确的被要求挂起的时候才会被挂起。

协同程序有点类似同步的多线程，在等待同一个线程锁的几个线程有点类似协同程序。

### 6.15.1 基本语法

协程的基本函数如表 6-10 所示。

表 6-10 协程的基本函数

方法	描述
<code>coroutine.create()</code>	创建 coroutine，返回 coroutine，参数是一个函数，当和 <code>resume</code> 配合使用的时候就唤醒函数调用
<code>coroutine.resume()</code>	重启 coroutine，和 <code>create</code> 配合使用
<code>coroutine.yield()</code>	挂起 coroutine，将 coroutine 设置为挂起状态，这个和 <code>resume</code> 配合使用能有很多有用的效果
<code>coroutine.status()</code>	查看 coroutine 的状态 注：coroutine 的状态有 3 种，即 <code>dead</code> 、 <code>suspend</code> 、 <code>running</code> ，具体什么时候有这样的状态请参考下面的程序
<code>coroutine.wrap()</code>	创建 coroutine，返回一个函数，一旦调用这个函数，就进入 coroutine，和 <code>create</code> 功能重复
<code>coroutine.running()</code>	返回正在跑的 coroutine，一个 coroutine 就是一个线程，当使用 <code>running</code> 的时候，返回一个 <code>corouting</code> 的线程号

以下实例演示了以上各个方法的用法：

```
-- coroutine_test.lua 文件
co = coroutine.create(
function(i)
print(i);
end
)

coroutine.resume(co, 1)      -- 1
print(coroutine.status(co))  -- dead

print("-----")

co = coroutine.wrap(
function(i)
print(i);
```

```

end
)

co(1)

print("-----")

co2 = coroutine.create(
function()
for i=1,10 do
    print(i)
    if i == 3 then
        print(coroutine.status(co2)) --running
        print(coroutine.running())    --thread:XXXXXX
    end
    coroutine.yield()
end
end
)

coroutine.resume(co2) --1
coroutine.resume(co2) --2
coroutine.resume(co2) --3

print(coroutine.status(co2)) -- suspended
print(coroutine.running())

```

```
print("-----")
```

以上代码的执行结果如下：

```

1
dead
-----
1
-----
1
2
3
running
thread: 0x7fb801c05868    false
suspended
thread: 0x7fb801c04c88    true
-----

```

通过 `coroutine.running` 可以看出，`coroutine` 在底层实现就是一个线程。当 `create` 一个 `coroutine` 的时候就是在新线程中注册了一个事件。当使用 `resume` 触发事件的时候，`create` 的 `coroutine` 函数就被执行了，当遇到 `yield` 的时候就代表挂起当前线程，等候再次 `resume` 触发事件。

接下来分析一个更详细的实例：

```

function foo (a)
print("foo 函数输出 ", a)

```

```

return coroutine.yield(2 * a) -- 返回 2*a 的值
end

co = coroutine.create(function (a, b)
print("第一次协同程序执行输出", a, b) -- co-body 1 10
local r = foo(a + 1)

print("第二次协同程序执行输出", r)
local r, s = coroutine.yield(a + b, a - b) -- a、b 的值为第一次调用协同程序时传入

print("第三次协同程序执行输出", r, s)
return b, "结束协同程序" -- b 的值为第二次调用协同程序时传入
end)

print("main", coroutine.resume(co, 1, 10)) -- true, 4
print("-- 分割线 ----")
print("main", coroutine.resume(co, "r")) -- true 11 -9
print("---- 分割线 ----")
print("main", coroutine.resume(co, "x", "y")) -- true 10 end
print("---- 分割线 ----")
print("main", coroutine.resume(co, "x", "y")) -- cannot resume dead coroutine
print("---- 分割线 ----")

```

以上代码的执行结果如下:

```

第一次协同程序执行输出    1    10
foo 函数输出    2
main    true    4
-- 分割线 ----
第二次协同程序执行输出r
main    true    11    -9
--- 分割线 ---
第三次协同程序执行输出    x    y
main    true    10    结束协同程序
--- 分割线 ---
main    false    cannot resume dead coroutine
--- 分割线 ---

```

实例解析:

- 调用 resume, 将协同程序唤醒, resume 操作成功返回 true, 否则返回 false。
- 协同程序运行。
- 运行到 yield 语句。
- yield 挂起协同程序, 第一次 resume 返回 (注意: 此处 yield 返回, 参数是 resume 的参数)。
- 第二次 resume, 再次唤醒协同程序 (注意: 此处 resume 的参数中, 除了第一个参数, 剩下的参数将作为 yield 的参数)。
- yield 返回。
- 协同程序继续运行。
- 如果使用的协同程序继续运行完成后继续调用 resume 方法则输出: cannot resume dead coroutine。



resume 和 yield 配合的强大之处在于, resume 处于主程中, 它将外部状态(数据)传入协同程序内部, 而 yield 则将内部的状态(数据)返回到主程中。

### 6.15.2 生产者 – 消费者问题

协程解决经典的生产者 – 消费者问题是非常简单的。下面实例演示对应方法:

```
local newProductor

function productor()
    local i = 0
    while true do
        i = i + 1
        send(i)          -- 将生产的物品发送给消费者
    end
end

function consumer()
    while true do
        local i = receive() -- 从生产者那里得到物品
        print(i)
    end
end

function receive()
    local status, value = coroutine.resume(newProductor)
    return value
end

function send(x)
    coroutine.yield(x)    -- x 表示需要发送的值, 值返回以后, 就挂起该协同程序
end

-- 启动程序
newProductor = coroutine.create(productor)
consumer()
```

以上代码的执行结果如下:

```
1
2
3
4
5
6
7
8
9
10
11
12
13
.....
```

## 6.16 Lua 错误处理

程序运行中进行错误处理是必要的,在进行文件操作、数据转移及 Web Service 调用过程中都会出现不可预期的错误,如果不注重错误信息的处理,就会造成信息泄漏,程序无法运行等情况。

任何程序语言中,都需要进行错误处理,Lua 的错误类型有语法错误、运行错误。下面将进行具体讲解。

### 6.16.1 语法错误

语法错误通常是由于对程序的组件(如运算符、表达式)使用不当引起的,下面是一个简单的实例:

```
-- test.lua 文件
a == 2
```

以上代码的执行结果如下:

```
lua: test.lua:2: syntax error near '=='
```

以上出现了语法错误,一个“=”号跟两个“=”号是有区别的。一个“=”是赋值表达式,两个“=”是比较运算。

另外一个实例:

```
for a= 1,10
  print(a)
end
```

执行以上程序会出现如下错误:

```
lua: test2.lua:2: 'do' expected near 'print'
```

语法错误比程序运行错误更简单,运行错误无法定位具体错误,而语法错误可以很快解决,如以上实例只要在 for 语句下面添加 do 即可:

```
for a= 1,10
do
  print(a)
end
```

### 6.16.2 运行错误

运行错误程序可以正常执行,但是会输出错误信息。如下面实例由于参数输入错误,程序执行时报错:

```
function add(a,b)
return a+b
end
```

```
add(10)
```

编译运行以下代码时，编译成功，但在运行的时候会产生如下错误：

```
lua: test2.lua:2: attempt to perform arithmetic on local 'b' (a nil value)
stack traceback:
  test2.lua:2: in function 'add'
  test2.lua:5: in main chunk
  [C]: ?
```

错误信息是由于程序缺少 `b` 参数引起的。

### 6.16.3 错误处理

可以使用 `assert` 和 `error` 两个参数处理错误，实例如下：

```
local function add(a,b)
  assert(type(a) == "number", "a 不是一个数字 ")
  assert(type(b) == "number", "b 不是一个数字 ")
  return a+b
end
add(10)
```

执行以上程序会出现如下错误：

```
lua: test.lua:3: b 不是一个数字
stack traceback:
  [C]: in function 'assert'
  test.lua:3: in local 'add'
  test.lua:6: in main chunk
  [C]: in ?
```

实例中 `assert` 首先检查第一个参数，若没问题，`assert` 不做任何事情；否则，`assert` 以第二个参数作为错误信息抛出。

### 6.16.4 error 函数

语法格式：

```
error (message [, level])
```

功能：终止正在执行的函数，并返回 `message` 的内容作为错误信息（`error` 函数永远都不会返回）。

通常情况下，`error` 会在 `message` 头部附加一些错误位置信息。

`Level` 参数指示获得错误的位置：

- `Level=1` [默认]：调用 `error` 位置（文件 + 行号）。
- `Level=2`：指出调用 `error` 函数的函数。
- `Level=0`：不添加错误位置信息。

### 6.16.5 pcall、xpcall、debug

在 Lua 中处理错误，可以使用函数 `pcall` (protected call) 包装需要执行的代码。`pcall` 接收一个函数和要传递给后者的参数，并执行。执行结果为有错误、无错误。返回值为 `true` 或 `false` 和 `errorinfo`。

语法格式如下：

```
if pcall(function_name, ...) then
-- 没有错误
else
-- 一些错误
end
```

简单实例：

```
> =pcall(function(i) print(i) end, 33)
33
true

> =pcall(function(i) print(i) error('error..') end, 33)
33
false          stdin:1: error..

>function f() return false,2 end
>if f() then print '1' else print '0' end
0
```

`pcall` 以一种“保护模式”调用第一个参数，因此 `pcall` 可以捕获函数执行中的任意错误。通常在错误发生时，希望获得更多的调试信息，而不只是发生错误的位置。但 `pcall` 返回时，已经销毁了调用栈的内容。

Lua 提供了 `xpcall` 函数，`xpcall` 接收第二个参数：一个错误处理函数。当错误发生时，Lua 会在调用栈查看引起错误的调用，所以，可以在这个函数中使用 `debug` 库获取关于错误的额外信息。

`debug` 库提供了两个通用的错误处理函数：

- `debug.debug`：提供一个 Lua 提示符，让用户检查错误原因。
- `debug.traceback`：根据调用栈构建扩展的错误消息。

上面例子的输出为

```
>=xpcall(function(i) print(i) error('error..') end, function() print(debug.
traceback()) end, 33) 33 stack traceback: stdin:1: in function [C]: in function
'error' stdin:1: in function [C]: in function 'xpcall' stdin:1: in main chunk [C]:
in ? false nil
```

`xpcall` 使用实例：

```
function myfunction ()
    n = n/nil
end
```

```
function myerrorhandler( err )
print( "ERROR:", err )
end
```

```
status = xpcall( myfunction, myerrorhandler )
print( status)
```

执行以上程序会出现如下错误:

```
ERROR:    test2.lua:2: attempt to perform arithmetic on global 'n' (a nil value)
false
```

## 6.17 Lua 调试

Lua 提供了 debug 库用于提供创建自定义调试器。Lua 本身没有内置的调试器,但很多开发者共享了 Lua 调试器代码。

Lua debug 库函数如表 6-11 所示。

表 6-11 Lua debug 库函数

序号	方法	说明
1	debug()	进入用户交互模式,运行用户输入的每个字符串。使用简单的命令以及其他调试设置,用户可以检阅全局变量和局部变量,改变变量的值,计算一些表达式等。输入一行仅包含 cont 的字符串将结束这个函数,这样调用者就可以继续向下运行
2	getfenv(object)	返回对象的环境变量
3	gethook(optional thread)	返回 3 个表示线程钩子设置的值:当前钩子函数、当前钩子掩码、当前钩子计数
4	getinfo ([thread,] f[,what])	返回关于一个函数信息的表。可以直接提供该函数,也可以用一个数字 f 表示该函数。数字 f 表示运行在指定线程的调用栈对应层次上的函数:0 层表示当前函数 (getinfo 自身);1 层表示调用 getinfo 的函数 (除非是尾调用,这种情况不计入栈);等等。如果 f 是一个比活动函数数量还大的数字, getinfo 返回 nil
5	debug.getlocal([thread,] f, local)	此函数返回在栈的 f 层处函数的索引为 local 的局部变量的名字和值。这个函数不仅用于访问显式定义的局部变量,也包括形参、临时变量等
6	getmetatable(value)	把给定索引指向的值的元表压入堆栈。如果索引无效,或这个值没有元表,函数将返回 0 并且不会向栈上压任何东西
7	getregistry()	返回注册表,这是一个预定义出来的表,可以用来保存任何 C 代码想保存的 Lua 值
8	getupvalue (f, up)	此函数返回函数 f 的第 up 个上值的名字和值。如果该函数没有那个上值,返回 nil。以“(” (开括号) 开头的变量名表示没有名字的变量 (去除了调试信息的代码块)
9	debug.sethook([thread,] hook, mask [, count])	将一个函数作为钩子函数设入。字符串 mask 以及数字 count 决定了钩子将在何时调用。掩码是由下列字符组合成的字符串,每个字符有其含义: "c": 每当 Lua 调用一个函数时,调用钩子; "r": 每当 Lua 从一个函数内返回时,调用钩子; "l": 每当 Lua 进入新的一行时,调用钩子
10	setlocal ([thread,] level, local, value)	这个函数将 value 赋给栈上第 level 层函数的第 local 个局部变量。如果没有那个变量,函数返回 nil。如果 level 越界,抛出一个错误

(续)

序号	方法	说明
11	setmetatable (value, table)	将 value 的元表设为 table (可以是 nil)。返回 value
12	setupvalue (f, up, value)	这个函数将 value 设为函数 f 的第 up 个上值。如果函数没有那个上值, 返回 nil, 否则, 返回该上值的名字
13	traceback ([thread,] [message [, level]])	如果有 message, 且不是字符串或 nil, 函数不做任何处理直接返回 message; 否则, 返回调用栈的栈回溯信息。字符串可选项 message 被添加在栈回溯信息的开头。数字可选项 level 指明从栈的哪一层开始回溯 (默认为 1, 即 traceback 调用处)

下面是一些 debug 库函数的应用实例:

```
function myfunction ()
print(debug.traceback("Stack trace"))
print(debug.getinfo(1))
print("Stack trace end")
return 10
end
myfunction ()
print(debug.getinfo(1))
```

以上代码的执行结果如下:

```
Stack trace
stack traceback:
  test2.lua:2: in function 'myfunction'
  test2.lua:8: in main chunk
  [C]: ?
table: 0054C6C8
Stack trace end
```

在以实例中, 用到了 debug 库的 traceback 函数和 getinfo 函数, getinfo 函数用于返回函数信息的表。

开发中经常需要调试函数内局部变量, 可以使用 getupvalue 函数设置这些局部变量, 实例如下:

```
function newCounter ()
local n = 0
local k = 0
return function ()
  k = n
  n = n + 1
return n
end
end

counter = newCounter ()
print(counter())
print(counter())

local i = 1
```

```

repeat
name, val = debug.getupvalue(counter, i)
if name then
    print ("index", i, name, "=", val)
    if(name == "n") then
        debug.setupvalue (counter,2,10)
    end
    i = i + 1
end -- if
until not name

print(counter())

```

以上代码的执行结果如下：

```

1
2
index    1    k    =    1
index    2    n    =    2
11

```

在以上实例中，计数器在每次调用时都会自动增加 1。实例中使用了 `getupvalue` 函数查看局部变量当前状态。可以将局部变量设置为新值。实例中，设置前 `n` 的值为 2，使用 `setupvalue` 函数将其设置为 10。现在调用函数，执行后输出结果为 11，而不是 3。

有两种调试类型可供选择。

- 命令行调试；
- 图形界面调试。

命令行调试器有 `RemDebug`、`clidebugger`、`ctrace`、`xdbLua`、`LuaInterface-Debugger`、`Rldb`、`ModDebug`。

图形界面调试器有 `SciTE`、`Decoda`、`ZeroBrane Studio`、`akdebugger`、`luaedit`。

可根据自身需求选择合适的调试类型。例如，我们用 `SciTE`，就直接用 `SciTE` 调试。

## 6.18 Lua 垃圾回收

Lua 采用自动内存管理机制。这意味着开发者不用操心新创建的对象需要的内存如何分配，也不用考虑对象不再被使用后怎样释放它们所占用的内存。Lua 通过运行垃圾收集器收集所有死对象（即在 Lua 中不可能再访问到的对象）来完成自动内存管理的工作。Lua 中所有用到的内存，如字符串、表、用户数据、函数、线程、内部结构等，都服从自动管理机制。

Lua 实现了一个增量标记——扫描收集器。它使用两个参数控制垃圾收集循环：垃圾收集器间歇率和垃圾收集器步进倍率。这两个参数都以百分数为单位（例如，值 100 在内部表示 1）。垃圾收集器间歇率控制着收集器需要在开启新的循环前要等待多久。增大这个值会减少收集器的积极性。当这个值比 100 小时，收集器在开启新的循环前不会等待。设置这

个值为 200 会让收集器等到总内存使用量达到之前的两倍时才开始新的循环。

垃圾收集器步进倍率控制着收集器运作速度相对于内存分配速度的倍率。增大这个值不仅会让收集器更加积极，还会增加每个增量步骤的长度。这个值不得小于 100，否则收集器就工作得太慢了以至于永远都做不完一个循环。默认值是 200，表示收集器以内存分配的“两倍”速工作。如果把步进倍率设为一个非常大的数字（比程序可能用到的字节数还大 10%），收集器的行为就像一个 stop-the-world 收集器。接着若把间歇率设为 200，收集器的行为就和过去的 Lua 版本一样了：每次 Lua 使用的内存翻倍时，就做一次完整的收集。

Lua 提供了函数 `collectgarbage` ([opt [, arg]]) 控制自动内存管理：

- `collectgarbage("collect")`：做一次完整的垃圾收集循环。通过参数 opt 提供了一组不同的功能。
- `collectgarbage("count")`：以 KB 为单位返回 Lua 使用的总内存数。这个值有小数部分，所以只需要乘上 1024 就能得到 Lua 使用的准确字节数（除非溢出）。
- `collectgarbage("restart")`：重启垃圾收集器的自动运行。
- `collectgarbage("setpause")`：将 arg 设为收集器的间歇率，返回间歇率的前一个值。
- `collectgarbage("setstepmul")`：返回步进倍率的前一个值。
- `collectgarbage("step")`：单步运行垃圾收集器。步长大小由 arg 控制。传入 0 时，收集器步进（不可分割的）一步。传入非 0 值时，收集器收集相当于 Lua 分配指定（KB）内存的工作。如果收集器结束一个循环将返回 true。
- `collectgarbage("stop")`：停止垃圾收集器的运行，调用重启前收集器只会因显式的调用运行。

以下代码演示了一个简单的垃圾回收实例：

```
mytable = {"apple", "orange", "banana"}
print(collectgarbage("count"))
mytable = nil

print(collectgarbage("count"))
print(collectgarbage("collect"))
print(collectgarbage("count"))
```

以上代码的执行结果如下（注意内存的变化）：

```
20.9560546875
20.9853515625
0
19.4111328125
```

## 6.19 Lua 面向对象

面向对象编程（Object Oriented Programming, OOP）是一种非常流行的计算机编程架构。以下几种编程语言都支持面向对象编程：C++、Java、Objective-C、Smalltalk、C#、



Ruby。

面向对象的主要特性如下：

- 1) 封装：能够把一个实体的信息、功能、响应都装入一个单独的对象中的特性。
- 2) 继承：继承的方法允许在不改动原程序的基础上对其进行扩充，这样使得原功能得以保存，而新功能也得以扩展。这有利于减少重复编码，提高软件的开发效率。
- 3) 多态：同一操作作用于不同的对象，可以有不同的解释，产生不同的执行结果。在运行时，可以通过指向基类的指针来调用实现派生类中的方法。
- 4) 抽象：抽象是简化复杂现实问题的途径，它可以为具体问题找到最恰当的类定义，并且可以在最恰当的继承级别解释问题。

### 6.19.1 Lua 中面向对象

对象由属性和方法组成。Lua 中最基本的结构是表，所以需要表描述对象属性，Lua 的函数用来表示方法，那么 Lua 中的类可以通过表 + 函数模拟出来。至于继承，可以通过元表模拟出来（不推荐使用，只模拟最基本的对象特性，通常情况下够用了）。

Lua 中的表在某种意义上是一种对象。像对象一样，表也有状态（成员变量），也有与对象的值独立的本性，特别是拥有两个不同值的对象代表两个不同的对象，一个对象在不同的时候也可以有不同的值，但它始终是一个对象。与对象类似，表的生命周期与由什么创建、在哪创建没有关系。对象有自己的成员函数，表也有：

```
Account = {balance = 0}
function Account.withdraw (v)
    Account.balance = Account.balance - v
end
```

这个定义创建了一个新的函数，并且保存在 Account 对象的 withdraw 域内，下面我们可以这样调用：

```
Account.withdraw(100.00)
```

下面是一个简单实例的例子。类包含了 3 个属性：area、length 和 breadth，printArea 方法用于打印计算结果：

```
-- Meta class
Rectangle = {area = 0, length = 0, breadth = 0}

-- 派生类的方法 new
function Rectangle:new (o,length,breadth)
    o = o or {}
    setmetatable(o, self)
    self.__index = self
    self.length = length or 0
    self.breadth = breadth or 0
    self.area = length*breadth;
    return o
```

```
end
```

```
-- 派生类的方法 printArea
function Rectangle:printArea ()
    print(" 矩形面积为 ",self.area)
end
```

### 1. 创建对象

创建对象是类实例分配内存的过程。每个类都有属于自己的内存并共享公共数据。

```
r = Rectangle:new(nil,10,20)
```

### 2. 访问属性

可以使用点号 (.) 访问类的属性：

```
print(r.length)
```

### 3. 访问成员函数

使用冒号 (:) 来访问类的属性：

```
r:printArea()
```

内存存在对象初始化时分配。

### 4. 函数重写

Lua 中可以重写基础类的函数，在派生类中定义自己的实现方式：

```
-- 派生类方法 printArea
function Square:printArea ()
    print(" 正方形面积 ",self.area)
end
```

### 5. Lua 类实例

下面是一个 Lua 类的完整实例：

```
-- Meta class
Shape = {area = 0}

-- 基础类方法 new
function Shape:new (o,side)
    o = o or {}
    setmetatable(o, self)
    self.__index = self
    side = side or 0
    self.area = side*side;
    return o
end

-- 基础类方法 printArea
function Shape:printArea ()
    print(" 面积为 ",self.area)
end
```

```
-- 创建对象
myshape = Shape:new(nil,10)
```

```
myshape:printArea()
```

以上代码的执行结果如下:

面积为      100

## 6.19.2 Lua 继承

继承是指一个对象直接使用另一对象的属性和方法,可用于扩展基础类的属性和方法。

下面代码演示了一个简单的继承实例:

```
-- Meta class
Shape = {area = 0}
-- 基础类方法 new
function Shape:new (o,side)
  o = o or {}
  setmetatable(o, self)
  self._index = self
  side = side or 0
  self.area = side*side;
  return o
end
-- 基础类方法 printArea
function Shape:printArea ()
  print(" 面积为 ",self.area)
end
```

Square 对象继承了 Shape 类的实例:

```
Square = Shape:new()
-- Derived class method new
function Square:new (o,side)
  o = o or Shape:new(o,side)
  setmetatable(o, self)
  self._index = self
  return o
end
```

以下实例演示继承一个简单的类,扩展派生类的方法,派生类中保留了继承类的成员变量和方法:

```
-- Meta class
Shape = {area = 0}
-- 基础类方法 new
function Shape:new (o,side)
  o = o or {}
  setmetatable(o, self)
  self._index = self
  side = side or 0
  self.area = side*side;
  return o
```

```

end
-- 基础类方法 printArea
function Shape:printArea ()
    print(" 面积为 ",self.area)
end

-- 创建对象
myshape = Shape:new(nil,10)
myshape:printArea()

Square = Shape:new()
-- 派生类方法 new
function Square:new (o,side)
    o = o or Shape:new(o,side)
    setmetatable(o, self)
    self.__index = self
    return o
end

-- 派生类方法 printArea
function Square:printArea ()
    print(" 正方形面积为 ",self.area)
end

-- 创建对象
mysquare = Square:new(nil,10)
mysquare:printArea()

Rectangle = Shape:new()
-- 派生类方法 new
function Rectangle:new (o,length,breadth)
    o = o or Shape:new(o)
    setmetatable(o, self)
    self.__index = self
    self.area = length * breadth
    return o
end

-- 派生类方法 printArea
function Rectangle:printArea ()
    print(" 矩形面积为 ",self.area)
end

-- 创建对象
myrectangle = Rectangle:new(nil,10,20)
myrectangle:printArea()

```

以上代码的执行结果如下：

```

面积为      100
正方形面积为    100
矩形面积为    200

```

## 6.20 Lua 数据库访问

本节主要介绍 Lua 5.1 中数据库的访问方法。在 Nginx 架构中，我们使用其他和 Nginx 架构整合更好的各种组件，本节仅用于学习通用 Lua 数据库访问方法。

通用数据访问使用的是 LuaSQL 库，LuaSQL 是一个开源库，支持的数据库有 ODBC、ADO、Oracle、MySQL、SQLite 和 PostgreSQL。本文为大家介绍 MySQL 的数据库连接。

LuaSQL 可以使用 LuaRocks 来安装，可以根据需要安装数据库驱动。

LuaRocks 安装方法：

```
$ wget http://luarocks.org/releases/luarocks-2.2.1.tar.gz
$ tar xzpf luarocks-2.2.1.tar.gz
$ cd luarocks-2.2.1
$ ./configure; sudo make bootstrap
$ sudo luarocks install luasocket
$ lua
Lua 5.3.0 Copyright (C) 1994-2015 Lua.org, PUC-Rio
> require "socket"
```

Windows 下安装 LuaRocks：<https://github.com/keplerproject/luarocks/wiki/Installation-instructions-for-Windows>。

安装不同数据库驱动：

```
luarocks install luasql-sqlite3
luarocks install luasql-postgres
luarocks install luasql-mysql
luarocks install luasql-sqlite
luarocks install luasql-odbc
```

也可以使用源码安装方式，Lua Github 源码地址：<https://github.com/keplerproject/luasql>。

Lua 连接 MySQL 数据库：

```
require "luasql.mysql"

-- 创建环境对象
env = luasql.mysql()

-- 连接数据库
conn = env:connect("数据库名", "用户名", "密码", "IP 地址", 端口)

-- 设置数据库的编码格式
conn:execute("SET NAMES UTF8")

-- 执行数据库操作
cur = conn:execute("select * from role")

row = cur:fetch({}, "a")

-- 创建文件对象
file = io.open("role.txt", "w+");
```

```

while row do
var = string.format("%d %s\n", row.id, row.name)
print(var)
file:write(var)
row = cur:fetch(row,"a")
end

file:close() -- 关闭文件对象
conn:close() -- 关闭数据库连接
env:close() -- 关闭数据库环境

```

## 6.21 小结

本章详细介绍了 Lua 的基本语法和数据类型；Lua 中最有特点的是表这个数据类型，大部分的复杂数据类型以及函数接口参数都可以使用表来实现；元表这个数据类型则提供了对两个表进行操作的能力；Lua 的库是以模块形式实现的。

本章还介绍了 Lua 中的协同程序，协程是 Lua 中比较特殊但是非常有效的多任务机制；Lua 中的错误处理机制和调试器接口对自我的 Lua 开发非常有用；介绍了 Lua 垃圾回收机制，Lua 中的资源都是自动管理内存的，用户不用管理；Lua 支持面向对象编程，可以使用表模拟类，使用元表实现继承。

本章最后介绍了 LuaSQL 库，使用户可以在通用 Lua 程序中访问多种数据库。

Lua 基础库和系统库在后面章节介绍，基础语法和基础库这两部分构成了 Lua 语言的全部内容。

# Lua 通用库

Lua 提供了一些通用库，使用者在 Lua 中可以直接进行字符串操作、表操作、文件操作、访问操作系统、进行科学计算等。本章汇总这些常用函数和操作，方便读者使用中查找。

## 7.1 字符串库

字符串操作在编程中应用广泛，特别是在互联网上，字符型流式数据、字符流式协议非常多，所以有大量的字符串操作，Lua 提供了很多的方法支持字符串操作。

下面是 Lua 内建的字符串函数。

- 1) `string.upper(argument)`：字符串全部转为大写字母。
- 2) `string.lower(argument)`：字符串全部转为小写字母。
- 3) `string.gsub(mainString, findString, replaceString, num)`：在字符串中替换。`mainString` 为要替换的字符串，`findString` 为被替换的字符，`replaceString` 为要替换的字符，`num` 为替换次数（可以忽略，则全部替换）。

例如：

```
> string.gsub("aaaa", "a", "z", 3);  
zzza    3
```

- 4) `string.find(str, substr, [init, [end]])`：在一个指定的目标字符串中搜索指定的内容（第三个参数为索引），返回其具体位置，不存在则返回 `nil`。

例如：

```
> string.find("Hello Lua user", "Lua", 1)
```

7 9

5) `string.reverse(arg)`: 字符串反转。

例如:

```
> string.reverse("Lua")
auL
```

6) `string.format(...)`: 返回一个类似 `printf` 的格式化字符串。

例如:

```
> string.format("the value is:%d",4)
the value is:4
```

7) `string.char(arg)` 和 `string.byte(arg[,int])`: `string.char(arg)` 用于将整型数字转化为字符并连接, `string.byte(arg[,int])` 用于将字符转换为整数值(可以指定某个字符,默认第一个字符)。

例如:

```
> string.char(97,98,99,100)
abcd
> string.byte("ABCD",4)
68
> string.byte("ABCD")
65
>
```

8) `string.len(arg)`: 计算字符串长度。

例如:

```
string.len("abc")
3
```

9) `string.rep(string, n)`: 将字符串 `string` 复制 `n` 次。

例如:

```
> string.rep("abcd",2)
abcdabcd
```

10) `..`: 连接两个字符串。

例如:

```
> print("www.google"..".com")
www.google.com
```

## 1. 字符串大小写转换

以下实例演示了如何对字符串大小写进行转换:

```
string1 = "Lua";
print(string.upper(string1))
print(string.lower(string1))
```



以上代码的执行结果如下：

```
LUA
lua
```

## 2. 字符串查找与反转

以下实例演示了如何对字符串进行查找与反转操作：

```
string = "Lua Tutorial"
-- 查找字符串
print(string.find(string, "Tutorial"))
reversedString = string.reverse(string)
print(" 新字符串为 ", reversedString)
```

以上代码的执行结果如下：

```
5      12
新字符串为      lairotuT auL
```

## 3. 字符串格式化

以下实例演示了如何对字符串进行格式化操作：

```
string1 = "Lua"
string2 = "Tutorial"
number1 = 10
number2 = 20
-- 基本字符串格式化
print(string.format(" 基本格式化 %s %s", string1, string2))
-- 日期格式化
date = 2; month = 1; year = 2014
print(string.format(" 日期格式化 %02d/%02d/%03d", date, month, year))
-- 十进制格式化
print(string.format("%.4f", 1/3))
```

以上代码的执行结果如下：

```
基本格式化 Lua Tutorial
日期格式化 02/01/2014
0.3333
```

## 4. 字符与整数相互转换

以下实例演示了字符与整数相互转换：

```
-- 字符转换
-- 转换第一个字符
print(string.byte("Lua"))
-- 转换第三个字符
print(string.byte("Lua", 3))
-- 转换末尾第一个字符
print(string.byte("Lua", -1))
-- 第二个字符
print(string.byte("Lua", 2))
-- 转换末尾第二个字符
print(string.byte("Lua", -2))
```

```
-- 整数 ASCII 码转换为字符
print(string.char(97))
```

以上代码的执行结果如下：

```
76
97
97
117
117
a
```

## 5. 其他常用字符串操作

以下实例演示了其他字符串操作，如计算字符串长度、字符串连接、字符串复制等。

```
string1 = "www."
string2 = "google"
string3 = ".com"
-- 使用 .. 进行字符串连接
print("连接字符串",string1..string2..string3)
```

```
-- 字符串长度
print("字符串长度",string.len(string2))
```

```
-- 字符串复制两次
repeatedString = string.rep(string2,2)
print(repeatedString)
```

以上代码的执行结果如下：

```
连接字符串      www.google.com
字符串长度      6
googlegoogle
```

## 7.2 表库

Lua 的表功能强大，能表示大部分的数据类型和对象类型，所以，编程中需要对表进行非常多的操作。Lua 内建了 Lua 表的操作函数，可以执行大部分的表操作，对于不支持的一些特殊性能，也提供了机制由开发者扩展。

下面列出 Table 表操作常用方法。

1) table.concat(table[,sep[,start[,end]]])：concat 是 concatenate（连锁、连接）的缩写。table.concat() 函数列出参数中指定 table 的数组部分从 start 位置到 end 位置的所有元素，元素间以指定的分隔符 (sep) 隔开。

2) table.insert(table[,pos[,value])：在 table 数组部分指定位置 (pos) 插入值为 value 的一个元素。pos 参数可选，默认为数组部分末尾。

3) table.maxn(table)：指定 table 中所有正数 key 值中最大的 key 值，如果不存在 key 值为正数的元素，则返回 0 (Lua 5.2 之后，该方法已经不存在了，本文使用了自定义函数实现)。

4) `table.remove(table[, pos])`: 返回 `table` 数组部分位于 `pos` 位置的元素。其后的元素会被前移, `pos` 参数可选, 默认为 `table` 长度, 即从最后一个元素删起。

5) `table.sort(table[, comp])`: 对给定的 `table` 进行升序排序。

下面是这几个方法的使用实例。

### 1. Table 连接

可以使用 `concat()` 方法连接两个 Table。

```
fruits = {"banana", "orange", "apple"}
-- 返回 table 连接后的字符串
print("连接后的字符串 ", table.concat(fruits))

-- 指定连接字符
print("连接后的字符串 ", table.concat(fruits, ", "))

-- 指定索引来连接 table
print("连接后的字符串 ", table.concat(fruits, ", ", 2, 3))
```

以上代码的执行结果如下:

```
连接后的字符串      bananaorangeapple
连接后的字符串      banana, orange, apple
连接后的字符串      orange, apple
```

### 2. Table 插入和移除

以下实例演示 Table 的插入和移除操作。

```
fruits = {"banana", "orange", "apple"}

-- 在末尾插入
table.insert(fruits, "mango")
print("索引为 4 的元素为 ", fruits[4])

-- 在索引为 2 的键处插入
table.insert(fruits, 2, "grapes")
print("索引为 2 的元素为 ", fruits[2])

print("最后一个元素为 ", fruits[5])
table.remove(fruits)
print("移除后最后一个元素为 ", fruits[5])
```

以上代码的执行结果如下:

```
索引为 4 的元素为      mango
索引为 2 的元素为      grapes
最后一个元素为          mango
移除后最后一个元素为    nil
```

### 3. Table 排序

以下实例演示 `sort()` 方法 (用于对 Table 进行排序) 的使用。

```
fruits = {"banana", "orange", "apple", "grapes"}
```

```

print(" 排序前 ")
for k,v in ipairs(fruits) do
    print(k,v)
end

table.sort(fruits)
print(" 排序后 ")
for k,v in ipairs(fruits) do
    print(k,v)
end

```

以上代码的执行结果如下：

排序前

```

1    banana
2    orange
3    apple
4    grapes

```

排序后

```

1    apple
2    banana
3    grapes
4    orange

```

#### 4. Table 最大值

在 Lua 5.2 之后，我们定义了 `table_maxn` 方法来实现 Table 的最大值。

以下实例演示了如何获取 Table 中的最大值。

```

function table_maxn(t)
    local mn = 0
    for k, v in pairs(t) do
        if mn < k then
            mn = k
        end
    end
    return mn
end

tbl = {[1] = "a", [2] = "b", [3] = "c", [26] = "z"}
print("tbl 长度 ", #tbl)
print("tbl 最大值 ", table_maxn(tbl))

```

以上代码的执行输出如下：

```

tbl 长度      3
tbl 最大值    26

```

## 7.3 文件 I/O 库

Lua 内建了文件 I/O 库支持对系统文件进行简单的读写操作。但是，需要注意的是，这些文件库功能简单，只能对文件进行打开和读写，不能操作目录、进行文件移动等（复杂的应用可参见后面介绍的 `lfs` 库，这里不再赘述）。另外，文件 I/O 库会阻塞进程，所以要慎

重使用，或合理使用技术将操作碎片化。

Lua I/O 库用于读取和处理文件，分为简单模式和完全模式。

- 简单模式：拥有一个当前输入文件和一个当前输出文件，并且提供针对这些文件相关的操作。
- 完全模式：使用外部文件句柄实现，即以面向对象的形式，将所有的文件操作定义为文件句柄。

简单模式较适用于一些简单的文件操作，但是在进行一些高级文件操作的时候，例如，同时读取多个文件这样的操作，使用完全模式则较为合适。

打开文件操作语句如下：

```
file = io.open (filename [, mode])
```

mode 的取值如下：

- r：以只读方式打开文件，该文件必须存在。
- w：打开只写文件，若文件存在则文件长度清为 0，即该文件内容会消失。若文件不存在则建立该文件。
- a：以附加的方式打开只写文件。若文件不存在，则会建立该文件，如果文件存在，写入的数据会被加到文件尾，即文件原先的内容会被保留（EOF 符保留）。
- r+：以可读写方式打开文件，该文件必须存在。
- w+：打开可读写文件，若文件存在则文件长度清为零，即该文件内容会消失。若文件不存在则建立该文件。
- a+：与 a 类似，但此文件可读可写。
- b：二进制模式，如果文件是二进制文件，可以加上 b。
- +：表示对文件既可以读也可以写。

### 7.3.1 简单模式

简单模式使用标准的 I/O 或使用一个当前输入文件和一个当前输出文件。

下面为 file.lua 文件代码，操作的文件为 test.lua（如果没有，需要创建该文件），代码如下：

```
-- 以只读方式打开文件
file = io.open("test.lua", "r")

-- 设置默认输入文件为 test.lua
io.input(file)

-- 输出文件第一行
print(io.read())

-- 关闭打开的文件
io.close(file)
```

```
-- 以附加的方式打开只写文件
file = io.open("test.lua", "a")

-- 设置默认输出文件为 test.lua
io.output(file)

-- 在文件最后一行添加 Lua 注释
io.write("-- test.lua 文件末尾注释 ")

-- 关闭打开的文件
io.close(file)
```

执行以上代码，会输出 test.ua 文件的第一行信息，并在该文件最后一行添加 lua 的注释。例如：

```
-- test.lua 文件末尾注释
```

上面实例中使用了 io. “x” 方法，其中 io.read() 没有带参数，参数可以是表 7-1 中的任意一个。

表 7-1 Lua 文件操作参数

模式	描述
"*n"	读取一个数字并返回它。例：file.read("*n")
"*a"	从当前位置读取整个文件。例：file.read("*a")
"*l" (默认)	读取下一行，在文件尾 (EOF) 处返回 nil。例：file.read("*l")
number	返回一个指定字符个数的字符串，或在 EOF 时返回 nil。例：file.read(5)

其他的 I/O 方法如下：

- io.tmpfile()：返回一个临时文件句柄，该文件以更新模式打开，程序结束时自动删除。
- io.type(file)：检测 obj 是否是一个可用的文件句柄。
- io.flush()：向文件写入缓冲中的所有数据。
- io.lines(optional file name)：返回一个迭代函数，每次调用将获得文件中的一行内容，当到文件尾时，将返回 nil，但不关闭文件。

7.3.2 完全模式

通常需要在同一时间处理多个文件时，可使用 file:function\_name 代替 io.function\_name 方法。以下实例演示如何同时处理同一个文件。

```
-- 以只读方式打开文件
file = io.open("test.lua", "r")

-- 输出文件第一行
print(file:read())

-- 关闭打开的文件
```

```

file:close()

-- 以附加的方式打开只写文件
file = io.open("test.lua", "a")

-- 在文件最后一行添加 Lua 注释
file:write("--test")

-- 关闭打开的文件
file:close()

```

执行以上代码会发现，结果输出了 test.lua 文件第一行信息，并在文件最后一行添加了 lua 注释，例如：

```

-- test.lua 文件

read 函数参数与简单模式一致。

```

### 7.3.3 其他方法

1) file:seek(optional whence, optional offset)：设置和获取当前文件位置，成功时返回文件位置（按字节），失败时返回 nil 与错误信息。

参数 whence 可以取以下值：

- "set"：从文件头开始。
- "cur"：从当前位置开始 [ 默认 ]。
- "end"：从文件尾开始。
- offset：默认为 0。

不带参数 file:seek() 表示返回当前位置。file:seek("set") 表示定位到文件头。file:seek("end") 表示定位到文件尾并返回文件大小。

2) file:flush()：向文件写入缓冲中的所有数据。

3) io.lines(optional filename)：以读模式打开 filename 文件并返回一个迭代函数，每次调用获得文件中的一行内容，读到文件尾时，返回 nil，并自动关闭文件。

若不带参数，io.lines() 函数的作用等同于 io.input():lines()，用于读取默认输入设备的内容，但结束时不关闭文件。例如：

```

for line in io.lines("main.lua") do
    print(line)
end

```

以下实例使用了 seek 方法，定位到文件倒数第 25 个位置并使用 read 方法的 \*a 参数，表示从当期位置（倒数第 25 个位置）读取整个文件。

```

-- 以只读方式打开文件
file = io.open("test.lua", "r")

file:seek("end", -25)

```

```
print(file:read("*a"))
```

```
-- 关闭打开的文件
file:close()
```

以上代码的执行结果如下：

st.lua 文件末尾 --test

7.4 数学库

在 Lua 编程中，会经常用到科学计算或工程计算，如科学计算、图形处理、界面特效处理等。需要进行科学计算的场合下可以使用 Lua 标准 math（数学库）进行复杂的数学运算。

表 7-2 列出 Lua Math 库函数。

表 7-2 Lua Math 库函数

序 列	函 数 名	描 述	示 例	结 果
1	abs(x)	取绝对值	math.abs(-100)	100
2	acos(x)	反余弦，返回角度	math.deg(math.acos(0.5))	60
3	asin(x)	反正弦，返回角度	math.deg(math.asin(0.5))	30
4	atan(x)	反正切，返回角度	math.deg(math.atan(1))	45
5	atan2(x, y)	反正切，返回角度	math.atan2(90.0, 45.0)	1.10714871
6	ceil(x)	向上取整	math.ceil(9.1)	10
7	cos(x)	余弦	math.cos(math.rad(60))	0.5
8	cosh(x)	双曲余弦	math.cosh(0.5)	1.276259652
9	deg(x)	弧度转角度	math.deg(math.pi)	180
10	exp(x)	e 的 x 次方	math.exp(4)	54.598150033144
11	floor(x)	向下取整	math.floor(9.9)	9
12	fmod(x,y)	取模运算	math.mod(14, 5)	4
13	frexp(x)	将参数拆成 x * (2 ^ y) 的形式	math.frexp(160)	0.625 8
14	huge	常量，代表无穷大	math.huge	inf
15	ldexp(x,y)	计算 x * (2 ^ y)	math.ldexp(0.625,8)	160
16	log(x)	计算 x 的自然对数	math.log(54.598150033144)	4
17	log10(x)	计算 10 为底，x 的对数	math.log10(1000)	3
18	max(x, ...)	取参数最大值	math.max(2,4,6,8)	8
19	min(x, ...)	取参数最小值	math.min(2,4,6,8)	2
20	mod(x,y)	取模	math.mod(65535,2)	1
21	modf(x)	取整数和小数部分	math.modf(20.12)	20 0.12
22	pi	圆周率	math.pi	3.1415926535898
23	pow(x,y)	计算 x 的 y 次幂	math.pow(2,16)	65536
24	randomseed(x)	设随机数种子	math.randomseed(os.time())	



(续)

序列	函数名	描述	示例	结果
25	random([m [, n]])	取随机数	math.random(5,90)	5 ~ 90
26	rad(x)	角度转弧度	math.rad(180)	3.1415926535898
27	sqrt(x)	开平方	math.sqrt(65536)	256
28	sin(x)	正弦	math.sin(math.rad(30))	0.5
29	sinh(x)	双曲正弦值	math.sinh(0.5)	0.5210953
30	tan(x)	正切	math.tan(math.rad(45))	1
31	tanh(x)	双曲正切值	math.tanh(0.5)	0.46211715

从函数名字可以看出，这些函数和 C、PHP、JavaScript 等语言中的数学函数是一致的，所以作用及使用场景也是一样的。函数本身并没有特殊功能，具体是由使用场景及数学计算决定的。

下面是一个三角函数的实例，可以看见 Lua 的数学函数定义和其他语言的是一样的。

```
radianVal = math.rad(math.pi / 2)

io.write(radianVal,"\n")
-- Sin value of 90(math.pi / 2) degrees
io.write(string.format("%.1f ", math.sin(radianVal)),"\n")-- Cos value of
90(math.pi / 2) degrees
io.write(string.format("%.1f ", math.cos(radianVal)),"\n")-- Tan value of
90(math.pi / 2) degrees
io.write(string.format("%.1f ", math.tan(radianVal)),"\n")-- Cosh value of
90(math.pi / 2) degrees
io.write(string.format("%.1f ", math.cosh(radianVal)),"\n")-- Pi Value in degrees
io.write(math.deg(math.pi),"\n")
```

以上代码的执行结果如下：

```
0.027415567780804
0.0
1.0
0.0
1.0
180
```

下面是一个普通数学函数的实例：

```
-- Floor
io.write("Floor of 10.5055 is ", math.floor(10.5055),"\n")-- Ceil
io.write("Ceil of 10.5055 is ", math.ceil(10.5055),"\n")-- Square root
io.write("Square root of 16 is ",math.sqrt(16),"\n")-- Power
io.write("10 power 2 is ",math.pow(10,2),"\n")
io.write("100 power 0.5 is ",math.pow(100,0.5),"\n")-- Absolute
io.write("Absolute value of -10 is ",math.abs(-10),"\n")--Random
math.randomseed(os.time())
io.write("Random number between 1 and 100 is ",math.random(),"\n")--Random
between 1 to 100
io.write("Random number between 1 and 100 is ",math.random(1,100),"\n")--Max
```

```
io.write("Maximum in the input array is ",math.max(1,100,101,99,999),"\n")--Min
io.write("Minimum in the input array is ",math.min(1,100,101,99,999),"\n")
```

以上代码的执行结果如下:

```
Floor of 10.5055 is 10
Ceil of 10.5055 is 11
Square root of 16 is 410
power 2 is 100100
power 0.5 is 10
Absolute value of -10 is 10
Random number between 1 and 100 is 0.22876674703207
Random number between 1 and 100 is 7
Maximum in the input array is 999
Minimum in the input array is 1
```

下面介绍一下 `math.random` 函数和 `math.randomseed` 函数。

`math.random` 是伪随机数生成函数, 在很多场合下需要使用。其支持的参数如下:

- 1) `math.random()`: 无参数, 生成 0 ~ 1 之间的浮点随机数。
- 2) `math.random(upper)`: 生成 1 ~ `upper` 之间的一个整数数值, `upper` 必须是整数。
- 3) `math.random(lower, upper)`: 生成 `lower` ~ `upper` 之间的一个整数数值。`lower` 和 `upper` 必须是整数。

下面是几个例子:

```
> = math.random()
0.0012512588885159
```

```
> = math.random()
0.56358531449324
```

```
> = math.random(100)
20
```

```
> = math.random(100)
81
```

```
> = math.random(70,80)
76
```

```
> = math.random(70,80)
75
```

`math.randomseed` 用于设置一个随机种子, 请看下面的例子:

```
math.randomseed()
> math.randomseed(1234)
> = math.random(), math.random(), math.random()
0.12414929654836      0.0065004425183874      0.3894466994232
> math.randomseed(1234)
> = math.random(), math.random(), math.random()
0.12414929654836      0.0065004425183874      0.3894466994232
```

对上面的例子分析可以发现：对于相同的随机种子，生成的随即序列一定是相同的。所以程序每次运行，赋予不同的种子就很重要。这时很自然想到使用系统时间作为随机种子，即

```
math.randomseed( os.time() )
```

但是在某些实时操作系统上，os.time() 精度是毫秒，这时 randomseed 工作正常，如果 os.time() 是秒级的系统，则随机数工作不正常，将会返回相同的随机数。

改进这个机制为

```
math.randomseed( tonumber(tostring(os.time()):reverse():sub(1,6)) )
```

就是把 time 返回的数值字符串倒过来（低位变高位），再取高位 6 位。这样，即使 time 变化很小，但是因为低位变了高位，种子数值变化也会很大，就可以使伪随机序列生成得更好一些。

所以，math.randomseed 描述上说是伪随机数生成器，使用上要注意这些限制。

## 7.5 操作系统库

应用开发中，会面临访问操作系统级功能的时候，如我们常用的获取系统时钟，通过系统环境变量传递信息，调用外部的工作执行一些 Lua 不方便执行的操作，用 curl 从指定的 RestFUL 接口读取数据等。Lua 为这些应用场景提供了操作系统库（OS 库）。

Lua 操作系统库函数如表 7-3 所示。

表 7-3 Lua 操作系统库函数

序号	库 / 方法	说明
1	os.clock ()	返回程序使用的 CPU 时间，单位为秒
2	os.date ([format [, time]])	返回日期和时间的字符串，可以通过 format 指定字符串格式
3	os.difftime (t2, t1)	返回 t1 和 t2 之间的秒数
4	os.execute ([command])	这个函数等同于 ANSI C 相同函数，向操作系统的 shell 传递命令以运行程序。如果成功，第一个返回值为 true，否则返回 nil
5	os.exit ([code [, close]])	调用 ANSI C 函数退出终端和主机程序。如果成功，返回码为 EXIT_SUCCESS；如果 code 是 false，返回码为 EXIT_FAILURE；如果 code 是数值，返回码将是这个数值
6	os.getenv (varname)	返回指定环境变量的值，如果未定义此变量，返回 nil
7	os.remove (filename)	删除文件（或空路径，在 POSIX 系统上）。如果失败，返回 nil，同时返回一个错误描述字符串
8	os.rename (oldname, newname)	将文件或路径改名为新的名字，如果函数失败，返回 nil，同时返回一个错误描述字符串
9	os.setlocale (locale [, category])	设置当前程序场所，场所是一个系统依赖的字符串。category 是一个可选的字符串，描述改变到哪个类别："all"、"collate"、"ctype"、"monetary"、"numeric" 或 "time"。默认类别是 "all"。函数成功则返回新的 locale，失败则返回 nil

(续)

序号	库 / 方法	说明
10	os.time ([table])	无参数调用时返回当前时间, 或者返回通过 table 指定的日期和时间代表的时间。表必须包含年、月、日、小时 (默认是 12)、分 (默认是 0)、秒 (默认是 0)、夏令时 (默认是 nil)
11	os.tmpname ()	返回一个可用于临时文件的文件名。这个文件必须可以打开或不用时删除

下面是一组简单函数操作示例:

```
-- Date with format
io.write("The date is ", os.date("%m/%d/%Y"), "\n")
-- Date and time
io.write("The date and time is ", os.date(), "\n")
-- Time
io.write("The OS time is ", os.time(), "\n")
-- Wait for some timefor i=1,1000000 doend
-- Time since Lua started
io.write("Lua started before ", os.clock(), "\n")
```

以上代码的执行结果如下:

```
The date is 01/25/2014
The date and time is 01/25/14 07:38:40
The OS time is 1390615720
Lua started before 0.013
```

上面是一些通用示例, 我们可以在需要的时候使用 OS 库, 通常更多地使用 os.time()、os.execute。

## 7.6 小结

本章介绍了字符串库、表库、文件 I/O 库和数学库。这些库都是常用库。字符串库用于操作字符串, 执行格式化、转换、合并等操作。表是 Lua 中功能强大的数据类型, 可以实现复杂的数据类型, 表库定义了表操作的函数。文件 I/O 库定义了对本地文件操作的基本接口, 复杂的文件操作需要使用其他扩展模块库实现, 本库只定义了基本的文件内容读写操作。数学库定义了类似于 C 数学库的全数学函数, 可以实现复杂的科学计算、客户端渲染、用户操作运算等操作。

### 第三部分 *Part 3*

# Nginx 开发技术

- 第 8 章 JSON 数据交换格式
- 第 9 章 nginx.conf 文件配置
- 第 10 章 Nginx 下 Lua 实现机制

## JSON 数据交换格式

JSON 是 JavaScript Object Notation (JavaScript 对象表示法) 的缩写, JSON 是一种存储和交换文本信息的语法, 类似 XML。JSON 比 XML 更小、更快, 更易解析, 是 JavaScript 内部使用的一种格式。JSON 因为其特点, 在互联网应用、嵌入式应用上得到了大量应用。我们在 Nginx 下的 Lua 开发中也大量使用 JSON 进行数据交换。本章介绍 JSON 这种常用的数据交换格式。

### 8.1 什么是 JSON

- JSON 指的是 JavaScript 对象表示法;
- JSON 是轻量级的文本数据交换格式;
- JSON 独立于语言;
- JSON 具有自我描述性, 更易理解。

JSON 使用 JavaScript 语法描述数据对象, 但是独立于语言和平台。JSON 解析器和 JSON 库支持许多不同的编程语言。目前非常多的动态 (PHP、JSP、.NET) 编程语言支持 JSON, 如 Lua、C++、Python 等, 主流编程语言也支持 JSON。

因为 JSON 具有诸多优点, 所以, Nginx 开发中大量使用 JSON 进行数据交换, OpenResty 中直接集成了 CJSON。JSON 和 Lua 表可以直接映射后使用, 后面的 Lua 开发中, 使用 JSON 描述和交换各种结构化数据、非结构化数据以及数据集更方便和直接。

下面是一个 JSON 实例:

```
{ "sites": [ { "name": "nginx" , "url": "www.nginx.org" }, { "name": "google" ,
```

```
"url":"www.google.com" }, { "name":"微博" , "url":"www.weibo.com" } ] }
```

其中，sites 对象是 1 个包含 3 个站点记录（对象）的数组，而数组中每个元素又可以包含自己的属性。总体来讲，JSON 是 key-value 式的结构，跟 NoSQL 非关系型数据和数据库有着天然的一致性。

## 8.2 JSON 转换为 JavaScript 对象

JSON 文本格式在语法上与创建 JavaScript 对象的代码相同。由于这种相似性，不需要解析器，JavaScript 程序能够使用内建的 eval() 函数，用 JSON 数据生成原生 JavaScript 对象。

本章着重描述数据封装和传输中用到的 JSON 特性。

## 8.3 JSON 与 XML 的比较

和 JSON 使用场景类似的还有 XML 格式，应用也非常广泛，本质上两者都是自描述性语言，复杂程度不同，各自有适应的场景。

两者的相似之处如下：

- 纯文本；
- 具有“自我描述性”（人类可读）；
- 具有层级结构（值中存在值）；
- 可通过 JavaScript 进行解析。

JSON 相较于 XML 不同之处如下：

- 没有结束标签；
- 更短；
- 读写速度更快；
- 能够使用内建的 JavaScript eval() 方法进行解析；
- 使用数组；
- 不使用保留字。

JSON 相对于 XML，数据冗余少，可以方便描述数据，编程简单，所以在数据交换过程中更有效。XML 也有自己的应用领域，适合描述数据之间的关系，也适合描述复杂的数据和关系，如用户的配置信息、权限信息等。JSON 适合透传数据库行和列信息网络协议、数据参数和返回结果等。

下面是一个基于 JSON 的网络协议示例：

```
{ "code" : 400, "command" : 7, "flow" : 1, "message" : "", "name" : "DDP",  
  "sequence" : 1, "session" : 1, "version" : "v1.0" }
```

上面的例子描述了一条协议的应答包，头和应答可以很好地自描述，调试方便，不用刻意编写协议转换代码，也便于网络上调试，协议数据与顺序无关。

下面是一个复杂带嵌套的数据返回示例：

```
{ "gateways": [ { "sensors": [ { "sensorId": "0101", "gps": "33.33333,22.2222", "region": "shunfan", "type": "HUMI", "name": "wow", "isOnline": null, "isAlarm": null } ], "gwId": "02", "gps": "33.22233,432.2233", "isAlarm": 1, "firmware": "", "interval": 10000, "region": "学校", "gwName": "you are 02" }, { "sensors": [ { "sensorId": "0101", "gps": "33.33333,22.2222", "region": "shunfan", "type": "HUMI", "name": "wow", "isOnline": null, "isAlarm": null } ], "gwId": "03", "gps": "33.22233,432.2233", "isAlarm": 1, "firmware": "", "interval": 10000, "region": "学校", "gwName": "you are 03" } ] }
```

JSON 对大型数据的传输相对比较节省流量，而且可以描述比较复杂的嵌套结构，且具备一定的可读性，适合在网络上传输。

## 8.4 JSON 语法规则

JSON 语法是 JavaScript 对象表示语法的子集。

- 数据在名称 / 值对中；
- 数据由逗号分隔；
- 花括号保存对象；
- 方括号保存数组。

### 1. JSON 名称 / 值对

JSON 数据的书写格式是“名称 / 值对”。名称 / 值对包括字段名称（在双引号中），后面写一个冒号，然后是值：

```
"name" : "Nginx"
```

这很容易理解，等价于下面的 JavaScript 语句：

```
name = "Nginx"
```

### 2. JSON 值

JSON 值可以是：

- 数字（整数或浮点数）；
- 字符串（在双引号中）；
- 逻辑值（true 或 false）；
- 数组（在方括号中）；
- 对象（在花括号中）；
- null。



### 3. JSON 对象

JSON 对象在花括号中书写，对象可以包含多个名称 / 值对：

```
{ "name": "nginx" , "url": "www.nginx.org" }
```

这一点也容易理解，与下面的 JavaScript 语句等价：

```
name = "nginx" url = "www.nginx.org"
```

### 4. JSON 数组

JSON 数组在方括号中书写，数组可包含多个对象：

```
{ "sites": [ { "name": "openresty" , "url": "openresty.org" }, { "name": "google" ,  
"url": "www.google.com" }, { "name": " 微博 " , "url": "www.weibo.com" } ] }
```

在上面的例子中，对象“sites”是包含 3 个对象的数组。每个对象代表一条关于网站（有名称和 URL）的记录。

### 5. JSON 文件

- JSON 文件的文件类型是 .json。
- JSON 文本的 MIME 类型是 application/json。

## 8.5 格式化

JSON 在具体的类实现代码里编码时有两种风格：格式化（style）和未格式化。未格式化风格不在每个元素后面输出换行符，例如：

```
[{"Name": "T1", "Value": "1"}, {"Name": "01H1", "Value": "96.2"}]
```

未格式化风格用于数据封装和传输。

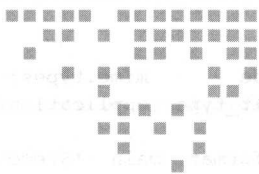
格式化风格在每个元素之后输出换行符，可读性更强，例如：

```
{  
  "idName": "01",  
  "name": "gateway1",  
  "typeName": "arduino",  
  "description": "test",  
  "isPublic": true  
}
```

格式化风格的 JSON 数据会导致接收代码处理复杂，因为换行符会使当前接收代码返回，因为协议接收端通常是以换行符作为一行数据接收完毕返回的判断机制，用户端需要自行拼包。因为我们通常将 JSON 用做数据交换，所以未格式化风格更适合在网络上传输。

## 8.6 小结

JSON 总体是一个 key-value 的结构，value 可以是数据，而数据可以嵌套。在 Nginx 的 Location 中，JSON 主要用于在各 Web 服务间提供数据交换的机制。各 Restful 服务以及 Location 多以 JSON 为数据交换格式，就连 PostgreSQL 返回的数据都是 JSON 的改良版本——BSON，支持二进制数据交换。



## nginx.conf 文件配置

nginx.conf 是 Nginx 的配置文件。Nginx 的工作流程是：在编译阶段选择要使用的模块并编译进整体工程中去。模块和业务的使用通过 nginx.conf 配置文件中配置指令的配置得以控制和实现，复杂的业务和自定义的业务逻辑使用 Lua 脚本实现。所以，nginx.conf 是我们开始开发及调整服务行为的首要途径。

配置正确的 nginx.conf 为我们提供一个正常运行且高效的服务框架，可以在框架内选择我们要介入的 HTTP 请求处理阶段，编写 Lua 代码提供具体实现。

### 9.1 默认 nginx.conf 文件

Nginx 提供了一个默认的 nginx.conf 模板，里面包含了一个 HTTP 服务配置块、一个 HTTPS 配置块和主要的全局配置项，我们可以在这个基础上修改从而形成需要的服务器配置文件。

配置文件内容如下：

```
#user nobody;  
worker_processes 1;  
  
#error_log logs/error.log;  
#error_log logs/error.log notice;  
#error_log logs/error.log info;  
  
#pid logs/nginx.pid;  
  
events {  
    worker_connections 1024;
```

```

}

http {
    include      mime.types;
    default_type application/octet-stream;

    #log_format  main  '$remote_addr - $remote_user [$time_local] "$request" '
    #              '$status $body_bytes_sent "$http_referer" '
    #              '"$http_user_agent"$http_x_forwarded_for"';

    #access_log  logs/access.log  main;

    sendfile      on;
    #tcp_nopush    on;

    #keepalive_timeout  0;
    keepalive_timeout  65;

    #gzip  on;

    server {
        listen      80;
        server_name  localhost;

        #charset koi8-r;

        #access_log  logs/host.access.log  main;

        location / {
            root      html;
            index      index.html index.htm;
        }

        #error_page  404              /404.html;

        # redirect server error pages to the static page /50x.html
        #
        error_page   500 502 503 504  /50x.html;
        location = /50x.html {
            root      html;
        }

        # proxy the PHP scripts to Apache listening on 127.0.0.1:80
        #
        #location ~ /\.php$ {
        #    proxy_pass  http://127.0.0.1;
        #}

        # pass the PHP scripts to FastCGI server listening on 127.0.0.1:9000
        #
        #location ~ /\.php$ {
        #    root          html;
        #    fastcgi_pass  127.0.0.1:9000;
        #    fastcgi_index index.php;
    }

```

```

#    fastcgi_param SCRIPT_FILENAME    /scripts$fastcgi_script_name;
#    include        fastcgi_params;
#}

# deny access to .htaccess files, if Apache's document root
# concurs with nginx's one
#
#location ~ /\.ht {
#    deny  all;
#}

}

# another virtual host using mix of IP-, name-, and port-based configuration
#
#server {
#    listen    8000;
#    listen    somename:8080;
#    server_name somename alias another.alias;

#    location / {
#        root    html;
#        index    index.html index.htm;
#    }
#}

# HTTPS server
#
#server {
#    listen    443 ssl;
#    server_name localhost;

#    ssl_certificate      cert.pem;
#    ssl_certificate_key  cert.key;

#    ssl_session_cache    shared:SSL:1m;
#    ssl_session_timeout  5m;

#    ssl_ciphers  HIGH:!aNULL:!MD5;
#    ssl_prefer_server_ciphers  on;

#    location / {
#        root    html;
#        index    index.html index.htm;
#    }
#}
}

```

对这个配置文件不用做任何修改，直接使用前面的命令启动 Nginx，就可以在 127.0.0.1 和实际 IP 上访问到默认的 index 页，这是一个标准的 Web 服务器。

## 9.2 nginx.conf 示例

下面给出一个实际使用的 nginx.conf 文件，这个文件实现了一个典型的应用，支持 MySQL 访问、Redis 访问、子链接访问等，内嵌了 Lua 代码。通过这个示例，读者可以对 nginx.conf 配置有一个总体印象。

```

user root;
worker_processes 4;
worker_rlimit_nofile 1000000;

error_log logs/error.log;
#error_log logs/error.log notice;
#error_log logs/error.log info;

#pid logs/nginx.pid;

events {
    use epoll;
    worker_connections 300000;
}

http {
    include mime.types;
    default_type text/html;

    #log_format main '$remote_addr - $remote_user [$time_local] "$request" '
    #                '$status $body_bytes_sent "$http_referer" '
    #                '"$http_user_agent"$http_x_forwarded_for";

    access_log off;
    server_tokens off;

    sendfile on;
    tcp_nopush on;
    tcp_nodelay on;
    open_file_cache max=10240 inactive=60s;
    open_file_cache_valid 80s;
    open_file_cache_min_uses 1;
    lua_shared_dict gkey 50m;
    lua_shared_dict gpost 10m;
    lua_shared_dict gvar 80m;
    lua_shared_dict msg_queue 300m;
    lua_shared_dict gsqs 100m;
    lua_shared_dict gex_session 50m;

    keepalive_timeout 0;
    #keepalive_timeout 600s;
    #keepalive_requests 10000;
    chunked_transfer_encoding off;
    #gzip on;
    lua_package_path "/usr/local/lib/lua/5.1/?.lua;;";

    upstream bk_mysql {

```

```

drizzle_server 10.185.220.120:3306 protocol=mysql dbname=test user=he
password=33Er3~#;
drizzle_keepalive max=300 overflow=reject mode=single;
}
upstream bk_master_db {
    drizzle_server 127.0.0.1:3306 protocol=mysql dbname=test user=he
password=33Er3~#;
    drizzle_keepalive max=100 overflow=reject mode=single;
}
upstream bk_redis {
    server 10.185.220.120:6009;
    # a pool with at most 1024 connections
    # and do not distinguish the servers:
    keepalive 1000;
}

upstream bk_svr_conf {
    server 10.195.194.47:9001;
    keepalive 1000;
}

server {
    listen          9500 default so_keepalive=on;
    server_name     10.185.194.47;
    set $pub_ip "120.26.57.240:9510";
    set $idm "121.40.249.246:8300";

    #charset koi8-r;
    charset utf-8;
    #chunked_transfer_encoding off;

    #access_log logs/host.access.log main;

    location /redis_set_ex {
        include /usr/local/ip_limit.conf;
        set $key $arg_key;
        set $expire $arg_expire;
        redis2_query set $key $request_body;
        redis2_query expire $key $expire;
        redis2_pass bk_redis;
    }

    location /redis_expire {
        include /usr/local/ip_limit.conf;
        set $key $arg_key;
        set $expire $arg_expire;
        redis2_query expire $key $expire;
        redis2_pass bk_redis;
    }

    location /redis_persist {
        include /usr/local/ip_limit.conf;
        set $key $arg_key;
        redis2_query persist $key;
        redis2_pass bk_redis;
    }
}

```

```

    }

    location /redis_set1 {
        include /usr/local/ip_limit.conf;
        set $key $arg_key;
        redis2_query set $key $request_body;
        redis2_query expire $key 86400;
        redis2_pass bk_redis;
    }

    location /redis_get {
        include /usr/local/ip_limit.conf;
        set $key $arg_key;
        redis2_query get $key;
        #redis2_query expire $key 86400;
        redis2_pass bk_redis;
    }

    location /redis_del {
        include /usr/local/ip_limit.conf;
        set $key $arg_key;
        redis2_query del $key;
        redis2_pass bk_redis;
    }

    location /mt_redis_set_ex {
        include /usr/local/ip_limit.conf;
        #access_by_lua_file access.lua;
        content_by_lua_block {
            local val=ngx.unescape_uri(ngx.var.arg_val)
            local resp = ngx.location.capture("/redis_set_ex?key=" .. ngx.
var.arg_key .. "&expire=" .. ngx.var.arg_expire, {
                method = ngx.HTTP_POST, body = val
            })
            ngx.exit(resp.status)
        }
    }

    location /user_status {
        include /usr/local/ip_limit.conf;
        default_type 'text/plain';
        lua_need_request_body on;
        client_max_body_size 50k;
        client_body_buffer_size 50k;
        content_by_lua "local srcid
if ngx.var.arg_uid == nil then
    ngx.exit(ngx.HTTP_INTERNAL_SERVER_ERROR)
else
    srcid=ngx.var.arg_uid
end
local gvar = ngx.shared.gvar
local user_status = gvar:get('user_status_' .. srcid)
if user_status == nil then
    ngx.print('0')
else

```



```

    ngx.print(user_status)
end
";
}

location /user_offline {
    include /usr/local/ip_limit.conf;
    default_type 'text/plain';
    lua_need_request_body on;
    client_max_body_size 50k;
    client_body_buffer_size 50k;
    content_by_lua "local srcid
if ngx.var.arg_uid == nil then
    local gvar = ngx.shared.gvar
    local gex_session = ngx.shared.gex_session
    local keys = gex_session:get_keys(0)
    local k,v
    for k, v in ipairs(keys) do
        gvar:set('user_session_' .. v, nil)
        gvar:set('user_status_' .. v, nil)
        gex_session:set(v,nil)
        local resp = ngx.location.capture('/redis_del?key=S_' .. v)
        ngx.say('cleanup session:' .. v)
    end
    ngx.exit(ngx.HTTP_OK)
else
    srcid=ngx.var.arg_uid
end
local gvar = ngx.shared.gvar
local session_key = gvar:get('user_session_' .. srcid)
if session_key ~= nil then
    gvar:set('user_session_' .. srcid, nil)
    local resp = ngx.location.capture('/redis_del?key=S_' .. srcid)
end
local user_status = gvar:get('user_status_'.. srcid)
if user_status ~= nil then
    gvar:set('user_status_' .. srcid, nil)
end
";
}

location /msg_count {
    include /usr/local/ip_limit.conf;
    default_type 'text/plain';
    lua_need_request_body on;
    client_max_body_size 50k;
    client_body_buffer_size 50k;
    content_by_lua "local srcid
if ngx.var.arg_uid == nil then
    ngx.exit(ngx.HTTP_INTERNAL_SERVER_ERROR)
else
    srcid=ngx.var.arg_uid
end
local msg_queue = ngx.shared.msg_queue
ngx.print('unread:' .. msg_queue:llen('user_msg_'.. srcid))

```

```

    ";
}
}

```

## 9.3 全局配置与顶层配置块

nginx.conf 从整体上讲分全局配置 (main)、顶层配置块及子配置块。放在配置文件中不用 {} 括起来的部分是全局配置, 第一层用 {} 括起来的是顶层配置块, 在顶层配置块中再用 {} 括起来的是子配置块。顶层配置块目前有 http、event、stream。下面分别描述顶层配置块和主要的子配置块。

### 9.3.1 main 全局配置

全局配置是 Nginx 在运行时与具体业务功能 (如 HTTP 服务或者 E-mail 服务代理) 无关的一些参数, 如工作进程数、运行的身份等。全局配置在配置文件最外层。

#### 1. 工作进程数

语法:

```
worker_processes number | auto;
```

默认:

```
worker_process 1;
```

配置块: 全局。

说明: 在配置文件的全局 main 部分, 管理进程接收任务并将请求分配给工作进程处理, 工作进程是实际的处理进程。工作进程的个数可以设置为 CPU 的核数 (grep ^processor /proc/cpuinfo | wc -l), 也可以是 auto 值, 如果开启了 ssl 和 gzip 更应该设置成与逻辑 CPU 数量一样甚至为 2 倍, 可以减少 I/O 操作。如果 Nginx 服务器还有其他服务, 可以考虑适当减少。

#### 2. 绑定工作进程到指定的 CPU 内核

语法:

```
worker_cpu_affinity cpumask[cpumask...]
```

默认: 无。

配置块: 全局。

说明: 如果 CPU 非常繁忙, 不一定会把每一个工作进程分配到一个核心上, 通过本指令手工指定会得到真正的并发 (仅对 Linux 有效)。例如:

```
worker_processes 4;
worker_cpu_affinity 1000 0100 0010 0001;
```

### 3. 工作进程最大打开文件数

语法:

```
worker_rlimit_nofile number;
```

默认: 无。

配置块: 全局。

说明: 改变工作进程最大打开文件数, 修改配置无须重启管理进程。

### 4. 工作进程的当前工作路径

语法:

```
worker_directory directory;
```

默认: 无。

配置块: 全局。

说明: 定义当前工作进程的工作路径, 主要用于生成 coredump 文件, 工作进程所在用户和组需要对工作目录有写权限。

### 5. 工作进程优先级

语法:

```
worker_priority number;
```

默认:

```
worker_priority 0;
```

配置块: 全局。

说明: 定义工作进程的优先级, 取值范围为  $-20 \sim +20$ 。

### 6. coredump 文件最大尺寸

语法:

```
worker_rlimit_core size;
```

默认: 无。

配置块: 全局。

说明: 定义工作进程 coredump 文件尺寸, 改动无须重启管理进程。

### 7. 是否以守护进程方式运行 Nginx

语法:

```
daemon on|off;
```

默认:

```
daemon on;
```

配置块: 全局。

说明：指明是否以守护进程方式运行 Nginx，默认为打开。

## 8. 是否以 master\_process 方式工作

语法：

```
master_process on|off;
```

默认：

```
master_process on;
```

配置块：全局。

说明：指明工作进程是否马上启动，主要用于 Nginx 深度开发使用，不是常规配置功能项。

## 9. error 日志设置

语法：

```
error_log /path/file level;
```

默认：

```
error_log logs/error.log err;
```

配置块：main、http、mail、stream、server、location。

说明：设置日志文件路径名，还可以设置要写入的错误级别。

## 10. 定义环境变量

语法：

```
env VAR|VAR=Value;
```

配置块：全局。

说明：直接设置操作系统上的环境变量。例如：

```
env MACCOC_OPTIONS;  
env PERL5LIB=/data/site/modules;  
env OPENSSL_ALLOW_PROXY_CERTS=1;
```

## 11. 引用其他配置文件

语法：

```
include /path/file;
```

默认：无。

配置块：any。

说明：文件名可以是绝对路径，也可以是相对路径，如果是相对路径，就是 nginx.conf 所在的路径。例如：

```
include mime.types;
```

```
include /usr/local/ip_limit.conf;
```

## 12. 锁定文件

语法:

```
lock_file file;
```

默认:

```
lock_file logs/nginx.lock;
```

配置块: 全局。

说明: Nginx 使用锁定文件实现 `accept_mutex`。在多数系统上, 这个锁用原子操作实现, 则这个值就被忽略掉了。这个配置在使用 lock file 机制的系统上使用。

## 13. 设置 pid 文件路径

语法:

```
pid path/file;
```

默认:

```
pid logs/nginx.pid;
```

配置块: main。

说明: 设置保存管理进程 ID 的文件, 用于使用进程 ID 操作 Nginx 的环境。

## 14. 设置工作进程运行时的用户及用户组

语法:

```
user username[groupname];
```

默认:

```
user nobody;
```

配置块: main。

说明: 指定工作进程工作时的用户和用户组, 主要在 Linux 和 UNIX 上使用。

## 15. SSL 硬件加速

语法:

```
ssl_engine device;
```

说明: 如果服务器上有 SSL 硬件加速设备, 可以通过配置实现硬件加速。可以使用 OpenSSL 命令查看是否有硬件设备:

```
openssl engine -t;
```

## 16. 工作进程中多线程读和写的线程池

语法:

```
thread_pool name threads=number [max_queue=number];
```

默认:

```
thread_pool default threads=32 max_queue=65535;
```

配置块: main。

说明: 这条指令从 1.7.11 版本出现, 定义工作进程中对文件读和写时用到的线程池。threads 参数定义线程池中线程的数量。max\_queue 限制允许在队列中等待的任务, 默认是 65 536 个任务。队列超出时, 任务将返回一个错误信息。

## 17. 工作进程时间频率

语法:

```
timer_resolution interval;
```

默认: 无。

配置块: main。

说明: 早期的 Linux 中, gettimeofday 是一个系统调用, 需要进行一次核心态和用户态的切换, 所以需要限制。现在的内核中, gettimeofday 仅是一次 vsyscall, 是对共享内存中的数据做访问, 代价不大, 所以目前通常不需要考虑这个问题。

## 9.3.2 events 配置块

events 模块中包含 Nginx 中所有处理连接的设置。events 是 Nginx 使用到的 I/O 事件模型, 是最主要的进出交互部分。Nginx 强大的部分就是在 Linux 上完美实现了 epoll 模型。

这里定义 events 的模型选择和参数。

常用配置项如下:

```
events{
    use epoll;
    worker_connections 20000;
}
```

### 1. 设置事件模型

语法:

```
use method;
```

默认: 无。

配置块: events。

说明: method 取值为 [kqueue | rtsig | epoll | /dev/poll | select | poll];。指令用于确定使用的事件模型, 一般在 Linux 下用 epoll。如果不设置本值, Nginx 会自动确定事件模型。

#### (1) 标准事件模型

select、poll 属于标准事件模型, 如果当前系统不存在更有效的方法, Nginx 会选择

select 或 poll。

## (2) 高效事件模型

kqueue：用于 FreeBSD 4.1+、OpenBSD 2.9+、NetBSD 2.0 和 MacOS X。使用双处理器的 MacOS X 系统使用 kqueue 可能会造成内核崩溃。

epoll：用于 Linux 内核 2.6 版本及以后的系统。

/dev/poll：用于 Solaris 7 11/99+、HP/UX 11.22+ (eventport)、IRIX 6.5.15+ 和 Tru64 UNIX 5.1A+。

eventport：用于 Solaris 10。为了防止出现内核崩溃的问题，有必要安装安全补丁程序。

查看 linux 版本号可以使用 cat /proc/version 命令。

## 2. 每个工作进程的最大连接数

语法：

```
worker_connections number;
```

默认：

```
worker_connections 512;
```

配置块：event。

说明：写在 events 部分，指每一个工作进程能并发处理（发起）的最大连接数（包含与客户端或后端被代理服务器间等所有连接数）。当 Nginx 作为反向代理服务器时，计算公式为最大连接数 = worker\_processes × worker\_connections/4，所以这里客户端最大连接数是 1024，这个可以增大到 8192，看情况而定，但不能超过 worker\_rlimit\_nofile。当 Nginx 作为 http 服务器时，以上计算公式里面改为除以 2。

## 3. 工作进程并发接收

语法：

```
multi_accept on|off;
```

默认：

```
multi_accept off;
```

配置块：events。

说明：如果 multi\_accept 是禁用的，一个工作进程同一时刻只能接收一个新的连接，否则，可以在同一时刻接收所有的连接。当类型是 kqueue 时，这个配置指令会被自动忽略。

## 4. AIO 最大输出数

语法：

```
worker_aio_requests number;
```

默认：

```
worker_aio_request 32;
```

配置块: events。

说明: 当在 epoll 连接处理方法中使用 AIO 时, 设置单工作进程 AIO 输出的最大数。

### 9.3.3 http 服务器配置块

HTTP 模块是 Nginx 中重要的模块, 顾名思义, 这是处理 HTTP 请求的模块。HTTP 模块中一般使用 HTTP 全局配置参数控制整体行为, 使用 server 配置虚拟主机, 包含监听地址、文档路径和各种 location。

反向代理、负载均衡等都是在内部的 server 等模块实现的。同时在各个子配置块或 location 等内部划分了许多阶段 (phase), 这些阶段可以注册 Lua 代码或 Lua 文件, 干预处理的过程。

本节重点讲述 HTTP 配置项的主要配置指令, 详细的配置请参阅官方 ngx\_http\_core\_module 模块介绍。

一个典型的 Web 服务器会包含全局配置、多个 server 块和多个 location 块:

```
http{
    gzip on;

    upstream{
        ...
    }
    ...
    server{
        listen localhost:80;
        ...
        location /webstatic {
            if ... {
                ...
            }
            root /opt/webresource;
            ...
        }
        location ~* \.(jpg|jpeg|png|jpe|gif){
            ...
        }
    }
    server{
        ...
    }
}
```

Nginx 为 Web 服务器提供了很多配置项, 这些配置项有的可以出现在任意一个配置块中, 有的只能在特定的块中, 这点在查看配置项描述时需要注意一下。

#### 1. 监听端口

语法:



```

listen address[:port] [default_server] [ssl] [http2 | spdy] [proxy_protocol]
[setfib=number] [fastopen=number] [backlog=number] [rcvbuf=size] [sndbuf=size]
[accept_filter=filter] [deferred] [bind] [ipv6only=on|off] [reuseport] [so_keepalive
=on|off][keepidle]:[keepintvl]:[keepcnt]];
listen port [default_server] [ssl] [http2 | spdy] [proxy_protocol] [setfib=number]
[fastopen=number] [backlog=number] [rcvbuf=size] [sndbuf=size] [accept_filter=filter]
[deferred] [bind] [ipv6only=on|off] [reuseport] [so_keepalive=on|off][keepidle]:[kee
pintvl]:[keepcnt]];
listen unix:path [default_server] [ssl] [http2 | spdy] [proxy_protocol]
[backlog=number] [rcvbuf=size] [sndbuf=size] [accept_filter=filter] [deferred] [bind]
[so_keepalive=on|off][keepidle]:[keepintvl]:[keepcnt]];

```

默认:

```
listen *:80 | *:8000;
```

配置块: server。

说明: listen 参数决定 Nginx 如何监听端口。在 listen 后面可以只加 IP、端口或主机名, 非常灵活。例如:

```

listen 127.0.0.1:8000;
listen 127.0.0.1;
listen 8000;
listen *:8000;
listen localhost:8000;

```

如果使用 IPv6, 那么可以这样使用:

```

listen [::]:8000;
listen [::1];

```

还可以加其他参数:

```
listen 127.0.0.1 default_server accept_filter=dataready backlog=1024;
```

主要参数的意义如下:

- default: 将所在的 server 块作为整个 Web 服务的默认 server 块。如果没有设置这个参数, 将以找到的第一个 server 块为默认 server 块。
- default\_server: 同 default。
- backlog=num: 表示 TCP 中 backlog 队列大小, 默认为 -1, 表示不设置。在 TCP 三次握手过程中, 进程还没有开始处理监听句柄, backlog 队列就会放置这些新连接。如果队列已满, 新的客户尝试连接, 则会失败。
- rcvbuf=size: 设置 so\_rcvbuf 接收缓冲区大小。
- sndbuf=size: 设置 so\_sndbuf 发送缓冲区大小。
- accept\_filter: 设置 accept 过滤器, 只对 FreeBSD 操作系统有用。
- deferred: 设置本参数后, 客户端建立连接, 并且完成了三次握手, 也不会调度工作进程来处理, 直到客户端实际请求数据到来才分配工作进程处理, 适用于大并发情况下减轻工作进程负担。

- bind: 绑定当前 ip/port 对, 只有在一个端口监听多个地址时才会生效。
- ssl: 在当前监听的端口上建立的连接必须使用 SSL 协议。

## 2. 主机名称

语法:

```
server_name name[...];
```

默认:

```
server_name "";
```

配置块: server。

说明: server\_name 后可以跟多个主机名称。例如:

```
server_name www.google.com mail.google.com;
```

## 3. location

语法:

```
location [=|~|~*|^~|@]/uri/{...}
```

配置块: server。

说明: location 尝试根据用户请求中的 URI 匹配上面的 /uri 表达式, 如果匹配, 就选择 location 中的配置处理用户请求。匹配的方式有很多种, 下面介绍匹配规则。

1) = 表示把 URI 作为字符串, 以便与参数中的 uri 做完全匹配。例如:

```
location = / {
# 只有当用户请求, 才会作用本 location 下的配置
...
}
```

2) ~ 表示匹配 URI 时是大小写敏感的。

3) ~\* 表示匹配 URI 时是大小写不敏感的。

4) ^~ 表示匹配时只需要前半部分与 URI 匹配即可。例如:

```
Location ^~ /image/{
# 以 /image/ 开始的请求都会匹配上
...
}
```

5) @ 表示仅用于 Nginx 服务内部请求之间重定向, 又名命名 location (named location)。可以在 URI 中使用正则表达式, 例如:

```
Location ~* \.(gif|jpg|jpeg)$ {
# 匹配以 .gif、.jpg、.jpeg 结尾的请求
}
```

location 是有顺序的, 当一个请求可以匹配多个 location 时, 只会被第一个匹配的 location 处理。

location 的匹配只能表达“如果匹配，则……”，如果需要匹配“如果不匹配，则……”，就比较难实现。可以在最后加一个 /location，如果前面都没有匹配上，则由“/”处理。

#### 4. 设置 root 路径

语法：

```
root path
```

默认：

```
html
```

配置块：http、server、location、if。

例如，定义资源文件相对于 HTTP 请求的根目录。

```
location /download/{
    root /opt/web/html/;
}
```

如果有一个 URI 是 /download/index/test.html，那么 Web 服务器会返回服务器上 /opt/web/html/download/index/test.html 文件的内容。

#### 5. 以别名方式设置资源路径

语法：

```
alias path;
```

配置块：location。

说明：alias 是用来设置文件资源路径的，与 root 的不同点在于如何解读 location 后面的 uri 参数，alias 和 root 会以不同的方式将用户请求映射到真正的磁盘文件上。例如，有一个请求的 URI 是 /conf/nginx.conf，而实际文件在 /usr/local/nginx/conf/nginx.conf，那么可以使用下面的方式设置：

```
location /conf{
    alias /usr/local/nginx/conf/;
}
```

如果用 root 设置，则为

```
location /conf{
    root /usr/local/nginx
}
```

alias 后面也可以添加正则表达式，例如：

```
location ~ ^/test/(\w+)\.(\w+)$ {
    alias /usr/local/nginx/$2/$1.$2;
}
```

在请求 /test/nginx.conf 时，会返回 /usr/local/nginx/conf/nginx.conf 文件的内容。

root 和 alias 配置块不同，root 使用更广。

## 6. 首页

语法:

```
index file ...;
```

默认:

```
index index.html;
```

配置块: http、server、location。

说明: 如果访问站点的 URI 是 /, 一般返回网站首页。index 后面可以跟多个参数, Nginx 按照顺序访问这些文件。例如:

```
location /{
    root path;
    index /index.html /html/index.php /index.asp;
}
```

## 7. 根据 HTTP 返回码重定向页面

语法:

```
error_page code[code...][|=answer-code]uri|@named_location
```

配置块: http、server、location、if。

说明: 当某个请求返回错误码时, 如果匹配上了 error\_page 中设置的页面, 则重定向到新的 URI 中。例如:

```
error_page 404          /404.html;
error_page 502 503 504  /50x.html;
error_page 403          http://example.com/forbidden.html;
error_page 404          = @fetch;
```

虽然重定向了 URI, 但返回的 HTTP 错误码还是原来的值, 可以使用 = 更改返回的错误码。例如:

```
error_page 404 =200 /empty.gif;
error_page 404 =403 /forbidden.gif;
```

如果不想修改 URI, 只想重定向到另外一个 location 中处理, 可以这样设置:

```
location / {
    error_page 404 @fallback;
}

location @fallback{
    proxy_pass http://backend;
}
```

其中, 404 请求会被反向代理到 http://backend 上游服务器中。

## 8. 是否允许递归使用 error\_page

语法:

```
recursive_error_pages[on|off];
```

默认:

```
recursive_error_pages off;
```

配置块: http、server、location。

说明: 标明是否允许递归定义 error\_page。

## 9. try\_files

语法:

```
try_files path1 [path2] uri;
```

配置块: server、location。

说明: try\_files 后要跟若干路径, 最后必须要有 uri 参数, 表示尝试按照顺序访问每一个 path, 如果可以有效地读取, 就直接向用户返回这个 path 对应的文件并结束请求, 否则继续向下访问。如果都找不到, 就定向到最后的 uri 上, 所以这个 uri 必须存在, 而且应该是可以有效重定向的。例如:

```
try_files /system/maintenance.html $uri $uri/index.html $uri.html @other;
location @other{
    proxy_pass http://backend;
}
```

还可以与 error\_page 配合使用。例如:

```
location / {
    try_files $uri $uri/ /error.php?c=404 =404;
}
```

## 10. HTTP 包体只存储到磁盘文件中

语法:

```
client_body_in_file_only on|clean|off;
```

默认:

```
client_body_in_file_only off;
```

配置块: http、server、location。

说明: 当值为非 off 时, 用户请求的 HTTP 包体都会存储到文件中, 即使只有 0 字节也会保存为文件。当请求结束时, 如果配置为 on, 这个文件不会被删除 (一般用来调试定位问题), 如果配置为 clean, 则会删除该文件。

## 11. HTTP 包体尽量写入缓冲区

语法:

```
client_body_in_single_buffer on|off;
```

默认:

```
client_body_in_single_buffer off;
```

配置块: http、server、location。

说明: 用户请求中的 HTTP 包体写入内存缓冲区中。当包体大小超过了 `client_body_buffer_size`, 还是会被写入文件中。

## 12. 存储 HTTP 头的缓冲区大小

语法:

```
client_header_buffer_size size;
```

默认:

```
client_header_buffer_size 1k;
```

配置块: http、server。

说明: 定义 HTTP 头缓冲区大小。有时, HTTP 头会超过这个大小, 这时 `large_client_header_buffers` 定义的缓冲区将生效。

## 13. 存储超大 HTTP 头部的缓冲区大小

语法:

```
large_client_header_buffers number size;
```

默认:

```
large_client_header_buffers 4 8k;
```

配置块: http、server。

说明: 定义超大 HTTP 头部缓冲区大小。如果 HTTP 请求行 (如 GET /index HTTP/1.1) 的大小超过了单个缓冲区个数, 会返回 “Request URI too large” (414)。请求中一般会有许多头域, 每一个头域大小也不能超过单个缓冲区大小, 否则会返回 “Bad request” (400)。请求行和请求头部总和不能超过缓冲区个数 × 缓冲区大小。

## 14. 存储 HTTP 包体的缓冲区大小

语法:

```
client_body_buffer_size size;
```

默认:

```
client_body_buffer_size 8k/16k;
```

配置块: http、server、location。

说明: 定义接收 HTTP 内存缓冲区大小。HTTP 包体会先接收到这块缓冲区里, 再决定是否写入磁盘。

## 15. HTTP 包体的临时存放目录

语法:

```
client_body_temp_path dir-path[level1[level2[level3]]];
```

默认:

```
client_body_temp_path client_body_temp;
```

配置块: http、server、location。

说明: 定义 HTTP 包体存放的临时目录。接收 HTTP 包体时, 如果包体的大小大于 `client_body_buffer_size`, 则会以一个递增的整数命名并存放于 `client_body_temp_path` 指定的目录中。后面跟着的 `level1`、`level2`、`level3`, 是为了防止一个目录下文件数量太多导致性能下降, 这样可以按照临时文件名最多再使用 3 层目录。例如:

```
client_body_temp_path /opt/nginx/client_temp 1 2;
```

如果新上传的 HTTP 包体使用 00000123456 作为临时文件名, 会被存放到这个目录中:

```
/opt/nginx/client_temp/6/45/00000123456
```

## 16. 连接内存池

语法:

```
connection_pool_size size;
```

默认:

```
connection_pool_size 256;
```

配置块: http、server。

说明: Nginx 对每个建立成功的 TCP 连接会预先分配一个内存池, 本参数指定内存池大小, 用于减少内核对于小块内存的分配次数。过大的 `size` 会占用更多的服务器内存, 更小的 `size` 则会引发更多的内存分配次数。

## 17. 请求池尺寸

语法:

```
request_pool_size size;
```

默认:

```
request_pool_size 4k;
```

配置块: http、server。

说明: Nginx 为每个 HTTP 请求分配一个内存池。TCP 连接关闭时会销毁 `connection_pool_size` 指定的连接内存池, HTTP 请求结束时会销毁 `request_pool_size` 指定的 HTTP 请求内存池。但它们创建、销毁时间不同, 因为一个 TCP 连接可能复用于多个 HTTP 请求。

## 18. HTTP 头读取超时时间

语法:

```
client_header_timeout time( 默认单位: 秒 );
```

默认:

```
client_header_timeout 60;
```

配置块: http、server、location。

说明: 客户端和服务端建立连接后开始接收 HTTP 头, 如果读取超时, 向客户端返回 408 (Request timed out) 错误。

## 19. 读取 HTTP 包体超时时间

语法:

```
client_body_time time;
```

默认:

```
client_body_time 60;
```

配置块: http、server、location。

说明: 读取包体的超时时间。

## 20. 发送响应的超时时间

语法:

```
send_timeout time;
```

默认:

```
send_timeout 60;
```

配置块: http、server、location。

说明: 发送响应的超时时间。超时发生时 Nginx 将关闭这个连接。

## 21. 重置超时连接

语法:

```
reset_timeout_connection on|off;
```

默认:

```
reset_timeout_connection off;
```

配置块: http、server、location。

说明: 连接超时后向客户端发送 RST 包重置连接。选项打开后, 在某个连接超时后, 不是使用正常的四次握手关闭 TCP 连接, 而是直接向客户端发送 RST 重置包, 不再等待用户应答, 直接释放套接字。相比正常关闭, 这种设置可以避免许多处于 FIN\_WAIT\_1、WAIT\_2、TIME\_WAIT 状态的连接。

因为使用 RST 重置连接会带来一些问题, 所以默认关闭。



## 22. 用户连接关闭方式

语法:

```
lingering_close off | on | always;
```

默认:

```
lingering_close on;
```

配置块: http、server、location。

说明: 设置 Nginx 关闭用户连接的方式。always 表示关闭之前必须先处理连接上所有用户发送的数据; off 表示不管数据直接关闭; on 是中间值, 一般都会处理完用户发送的数据, 除非业务上认为这些数据是不必要的。

## 23. 用户连接关闭时间

语法:

```
lingering_time time;
```

默认:

```
lingering_time 30s;
```

配置块: http、server、location。

说明: lingering\_close 启用后, 这个配置项对于上传大文件很有用。当用户请求的 content-length 大于 max\_client\_body\_size 时, Nginx 会立即返回 413 (request entity too large) 响应, 但是很多客户端可能不处理 413 返回值, 仍然上传数据。这时, 经过 lingering\_time 后, Nginx 不管是否还有数据在上传, 直接把这个连接关掉。

## 24. 用户连接关闭超时值

语法:

```
lingering_timeout 5s;
```

配置块: http、server、location。

说明: lingering\_close 生效后, 检查关闭连接前是否有用户发送的数据到达服务器。如果超过 lingering\_timeout 时间后还没有数据可读, 就直接关闭连接; 否则, 必须读取完连接缓冲区上的数据并丢弃后才会关闭连接。

## 25. keepalive 超时时间

语法:

```
keepalive_timeout time;
```

默认:

```
keepalive_timeout 75;
```

配置块: http、server、location。

说明：一个 *keepalive* 连接在闲置超过一定时间后，服务器和浏览器都会关闭这个连接。这个值是用于限制 Nginx 服务器的，Nginx 会把这个值传给浏览器，但每个浏览器对待 *keepalive* 的策略可能是不同的。

## 26. *keepalive* 连接上最大承载数

语法：

```
keepalive_requests n;
```

默认：

```
keepalive_requests 100;
```

配置块：http、server、location。

说明：一个 *keepalive* 长连接上默认最多只能发送 100 个请求。

## 27. tcp 尼古拉算法开关

语法：

```
tcp_nodelay on|off;
```

默认：

```
tcp_nodelay on;
```

配置块：http、server、location。

说明：确定对 *keepalive* 连接是否使用 *tcp\_nodelay* 选项。

## 28. tcp 协议 *nopush* 开关

语法：

```
tcp_nopush on|off;
```

默认：

```
tcp_nopush off;
```

配置块：http、server、location。

说明：在打开 *sendfile* 选项时，确定是否开启 FreeBSD 系统上的 *tcp\_nopush* 或 Linux 系统上的 *tcp\_cork* 功能。打开 *tcp\_nopush* 后，将会在发送响应时把整个响应包头放到一个 TCP 包中发送。

## 29. MIME type 到文件扩展名的映射

语法：

```
type{...};
```

配置块：http、server、location。

说明：定义 MIME type 到文件扩展名的映射。多个扩展名可以映射到同一个 MIME

type。例如：

```
types{
    text/html      html;
    text/html      conf;
    image/gif      gif;
    image/jpeg     jpg;
}
```

### 30. 默认 MIME type

语法：

```
default_type MIME-type;
```

默认：

```
default_type text/plain;
```

配置块：http、server、location。

说明：当找不到 MIME type 的映射时，使用默认的 MIME type 作为 HTTP 头域中的 content-type。

### 31. MIME type 映射散列桶内存大小

语法：

```
types_hash_bucket_size size;
```

默认：

```
types_hash_bucket_size 32|64|128;
```

配置块：http、server、location。

说明：Nginx 使用散列表存储 MIME type 映射，本指令定义了每个散列桶占用内存的大小。

### 32. MIME type 映射散列桶大小

语法：

```
types_hash_max_size size;
```

默认：

```
types_hash_max_size 1024;
```

配置块：http、server、location。

说明：本配置影响散列表的冲突率。值越大，就消耗越多的内存，但冲突率越小，检索速度更快。值越小，冲突机率高，检索效率降低，但节省内存。

### 33. 按 HTTP 方法限制用户请求

语法：

```
limit_except method ... {...}
```

配置块: location。

说明: Nginx 通过 limit\_except 后面指定的方法名限制用户请求。方法名取值包括 GET、HEAD、POST、PUT、DELETE、MKCOL、COPY、MOVE、OPTIONS、PROPFIND、PROPPATCH、LOCK、UNLOCK、PATCH。例如:

```
limit_except GET{
    allow 192.168.1.0/32;
    deny all;
}
```

上面方法中, 允许 GET 方法和 HEAD 方法通过, 其他方法禁止。

### 34. HTTP 请求包体最大值

语法:

```
client_max_body_size size;
```

默认:

```
client_max_body_size 1m;
```

配置块: http、server、location。

说明: 浏览器在发送较大包体的请求时, 会在头部带一个 content-type 字段, client\_max\_body\_size 是用来限制 content-type 大小的, 这就使得 Nginx 不用等待至接收完所有的包体, 会大大节约时间。例如, 当用户上传一个 1GB 的文件时, Nginx 接收完包头后发现包体超大, 直接发送 413 (request entity too large) 响应给客户端。

### 35. 对请求限速

语法:

```
limit_rate speed;
```

默认:

```
limit_rate 0;
```

配置块: http、server、location、if。

说明: 限制客户端每秒传输字节数。默认为 0, 表示不限速。针对不同客户端, 可以用 \$limit\_rate 参数执行不同的限速策略。例如:

```
server{
    if($slow)
        set $limit_rate 4k;
}
```

### 36. 限速阈值

语法:

```
limit_rate_after time;
```

默认:

```
limit_rate_after 1m;
```

配置块: http、server、location、if。

说明: 此配置表示 Nginx 向客户端发送的响应长度超过 limit\_rate\_after 后才开始限速。

例如:

```
limit_rate_after 1;
limit_rate 100k;
```

### 37. sendfile 系统调用

语法:

```
sendfile on|off;
```

默认:

```
sendfile off;
```

配置块: http、server、location。

说明: 可以启用 Linux 上的 sendfile 系统调用发送文件, 它减少了用户态与核心态之间的两次内存复制, 可以从磁盘中读取文件后直接在内核态发送到网卡, 提高了效率。

### 38. AIO 系统调用

语法:

```
aio on|off;
```

默认:

```
aio off;
```

配置块: http、server、location。

说明: 表示是否在 FreeBSD 或 Linux 系统上启用内核级别的异步文件 I/O 功能, AIO 与 sendfile 是互斥的。

### 39. direction 选项

语法:

```
directio size|off;
```

默认:

```
directio off;
```

配置块: http、server、location。

说明: 在 FreeBSD 或 Linux 系统上使用 O\_DIRECT 选项读取文件, 缓冲区大小为 size, 对大文件读取速度有优化作用, 与 sendfile 互斥。

#### 40. directio 选项对齐尺寸

语法:

```
directio_alignment size;
```

默认:

```
directio_alignment 512;
```

配置块: http、server、location。

说明: 与 directio 一起使用, 指定以 directio 方式读取文件时的对齐方式。一般情况下, 512B 足够了, 但对一些高性能文件系统, 如 Linux 下的 XFS 文件系统, 可能需要设置到 4KB 作为对齐方式。

#### 41. 打开文件缓存

语法:

```
open_file_cache max=N[inactive=time]|off;
```

默认:

```
open_file_cache off;
```

配置块: http、server、location。

说明: 文件缓存会在内存中存储 3 种信息。

- 文件句柄、文件大小、上次修改时间;
- 已经打开过的目录结构;
- 没有找到的或者没有权限操作的文件信息。

通过读取缓存减少了磁盘操作。

3 个参数如下:

- max: 表示在内存中存储元素的最大个数, 当达到最大限制时, 采用 LRU 算法从缓存中淘汰最近最少使用的元素。
- inactive: 表示在 inactive 指定的时间段内没有被访问过的元素将会被淘汰。默认时间为 60 秒。
- off: 关闭缓存功能。

例如:

```
open_file_cache max=1000 inactive=20s
```

#### 42. 是否缓存打开文件时的错误信息

语法:

```
open_file_cache_errors on|off;
```

默认:

```
open_file_cache_errors off;
```

配置块: http、server、location。

说明: 指定是否缓存打开文件时的错误信息。

### 43. 不被淘汰的最小访问次数

语法:

```
open_file_cache_min_uses number;
```

默认:

```
open_file_cache_min_uses 1;
```

配置块: http、server、location。

说明: 本参数与 `open_file_cache_min_uses` 配合使用。如果在 `inactive` 指定的时间段内, 访问次数超过了 `open_file_cache_min_uses` 指定的最小次数, 那么将不会被淘汰出缓存。

### 44. 检验缓存中元素有效率性的频率

语法:

```
open_file_cache_valid time;
```

默认:

```
open_file_cache_valid 60;
```

配置块: http、server、location。

说明: 默认每 60 秒检查一次缓存中的元素是否有效。

### 45. 忽略不合法的 HTTP 头部

语法:

```
ignore_invalid_headers on|off;
```

默认:

```
ignore_invalid_headers on;
```

配置块: http、server。

说明: on 情况下, 当出现不合法的 HTTP 头部时, Nginx 会忽略此 HTTP 头部; off 情况下则会拒绝服务, 并向用户发送 400 (bad request) 错误。

### 46. HTTP 头部是否允许下划线

语法:

```
underscores_in_headers on|off;
```

默认:

```
underscores_in_headers off;
```

配置块: http、server。

说明: 默认为 off, 表示头域名称不允许带下划线。

#### 47. if\_Modified\_Since 头域处理策略

语法:

```
if_Modified_Since[off|exact|before];
```

默认:

```
if_modified_since exact;
```

配置块: http、server、location。

说明: 为了更好的性能, 客户端浏览器会在客户端本地缓存一些文件, 并保存当时获取的时间, 下次向服务器获取缓存过的资源时, 就携带 If-Modified-Since 头域, 本指令根据参数决定如何处理。

- off: 忽略 If-Modified-Since 头域, 每次都读取文件给客户端。
- exact: 将头域时间和文件修改时间做精确比较, 如果未匹配上, 则读取文件返回给用户; 如果匹配上, 则表示浏览器本地为最新内容, 返回 304 (not modified) 给客户端, 直接使用本地缓存, 节省带宽。
- before: 比 exact 更宽松, 只要文件修改时间等于或早于 If-Modified-Since 头部时间, 就向客户端返回 304 (not modified)。

#### 48. 文件未找到时是否记录到 error 中

语法:

```
log_not_found on|off;
```

语法:

```
log_not_found on;
```

配置块: http、server、location。

说明: 当处理用户文件请求时, 如果没找到文件是否记录到 error.log 中, 一般用于调试定位问题, 生产系统需要关掉。

#### 49. 合并相邻斜线

语法:

```
merge_slashes on|off;
```

默认:

```
merge_slashed on;
```

配置块: http、server、location。

说明: 指明是否合并相邻的 /, 例如, 如果 //test///a.txt 为 on, 则会匹配为 location /



test/a.txt; 如果为 off, 则不会匹配, URI 还是 //test///a.txt。

## 50. DNS 解析

语法:

```
resolver address ...;
```

配置块: http、server、location。

说明: 设置 DNS 域名解析服务器地址。例如:

```
resolver 127.0.0.1 192.0.2.1;
```

## 51. DNS 解析超时时间

语法:

```
resolver_timeout time;
```

默认:

```
resolver_timeout 30s;
```

配置块: http、server、location。

说明: 表示 DNS 解析的超时时间。

## 52. 返回错误页面时是否注明 Nginx 版本

语法:

```
server_token on|off;
```

默认:

```
server_token on;
```

配置块: http、server、location。

说明: 表明处理出错的请求时, 是否在应答头域的 server 域内标明 Nginx 版本, 用于定位问题。

## 53. upstream 块

语法:

```
upstream name{...}
```

配置块: http。

说明: upstream 块定义一个上游服务器集群, 用于反向代理中的 proxy\_pass 指令。

例如:

```
upstream backend{
    server backend1.example.com;
    server backend2.example.com;
    server backend3.example.com;
}
```

```
server{
    location / {
        proxy_pass http://backend;
    }
}
```

## 54. 上游服务器名

语法:

```
server name[parameters];
```

配置块: upstream。

说明: server 配置项指定了一台上游服务器名, 可以是域名、IP 地址、UNIX 句柄等。

参数如下:

- weight=number: 这台上游服务器的权重, 默认为 1。
- max\_fails=number: 与 fail\_timeout 配合使用, 指在 fail\_timeout 时间段内, 如果这台上游服务器转发失败次数超过 number, 则认为在 fail\_timeout 内该服务器不可用。默认为 1, 0 表示不检查失败次数。
- fail\_timeout=time: 表示该时间段内上游服务器转发失败多少次后就认为该服务器暂时不可用, 用于优化反向代理。默认为 10 秒。这个超时不是连接、发送等超时。
- down: 表示该服务器永久下线, 只在使用 ip\_hash 配置项时才有用。
- backup: 使用 ip\_hash 无效。表示该上游服务器仅是备份服务器, 只有在所有非备份服务器都失效后, 才会向备份服务器转发请求。例如:

```
upstream backend{
    server backend1.example.com weight=5;
    server 127.0.0.1 max_fails=3 fail_timeout=30s;
    server unix:/tmp/backend3;
}
```

## 55. 基于 IP 的 hash 算法

语法:

```
ip_hash;
```

配置块: upstream。

说明: 在如有会话的 Web 请求下, 如 Java、PHP 的动态页面, 希望一个客户端的请求最好始终分配到固定的一台上游服务器中, 这样就可以保持会话, 而不用在上游服务器中做会话同步。ip\_hash 就是解决这个问题的, 根据客户端 IP Hash 出一个 key, 将 key 与 upstream 集群中的上游服务器数量取模, 然后以取模后的结果把请求转发到对应的上游服务器上, 确保同一个客户端的请求只转发到指定的上游服务器中。

ip\_hash 与 weight 配置不可同时使用。如果 upstream 中的一台服务器暂时不可用, 不能直接删除该服务器配置, 要使用 down 标志, 否则转发策略会混乱。例如:

```

upstream backend{
    ip_hash;
    server backend1.example.com;
    server backend2.example.com;
    server backend3.example.com down;
    server backend4.example.com;
}

```

## 56. 转发当前代理

语法:

```
proxy_pass URL;
```

配置块: location、if。

说明: 这个配置将当前请求反向代理到 URL 指定的服务器上, URL 可以是主机名或 IP 地址 + 端口。例如:

```
proxy_pass http://localhost:8000/uri/;
```

也可以是 UNIX 句柄:

```
proxy_pass http://unix:/path/to/backend.socket:/uri/;
```

也可以直接使用 upstream 块:

```

upstream backend{
    .....
}

server{
    location / {
        proxy_pass http://backend;
    }
}

```

也可以把 HTTP 转换成 HTTPS:

```
proxy_pass https://10.12.0.1;
```

默认情况下, 反向代理不会转发请求中的 Host 头部, 如果需要转发, 那么需要加上配置项:

```
proxy_set_header Host $host;
```

## 57. 转发使用的协议方法

语法:

```
proxy_method method
```

配置块: http、server、location。

说明: 设置转发时的协议方法名。例如:

```
proxy_method POST;
```

则客户端发来的 GET 请求在转发时方法名也会改为 POST。

### 58. 转发时隐藏的头域

语法:

```
proxy_hide_header the_header;
```

配置块: http、server、location。

说明: Nginx 会将上游服务器的响应转发给客户端, 但默认不会转发以下头域——Date、Server、X-Pad、X-Accel-\*。使用 proxy\_hide\_header 后, 可以任意指定哪些头域不用转发。例如:

```
proxy_hide_header Cache-Control;
proxy_hide_header MicrosoftOffieWebServer;
```

### 59. 转发时明确使用的头域

语法:

```
proxy_pass_header the_header;
```

配置块: http、server、location。

说明: 与 proxy\_hide\_header 功能相反, 可以将原来禁止的头域进行转发。

### 60. 是否转发请求包体

语法:

```
request_pass_request_body on|off;
```

默认:

```
request_pass_request_body on;
```

配置块: http、server、location。

说明: 指定是否向上游服务器转发请求包体。

### 61. 是否转发请求头

语法:

```
proxy_pass_request_header on|off;
```

默认:

```
proxy_pass_request_header on;
```

配置块: http、server、location。

说明: 确定是否向上游服务器转发请求包头。

### 62. 重定向转发

语法:

```
proxy_redirect[default|off|redirect replacement];
```

默认:

```
proxy_redirect default;
```

配置块: http、server、location。

说明: 当上游服务器返回的响应是重定向或刷新请求 (301、302) 时, proxy\_redirect 可以重设 HTTP 头部的 location 或 refresh 字段。例如:

```
proxy_redirect http://localhost:8000/two/ http://frontend/one/;
```

则上游的 location 字段就被替换了, 客户端的一致性检查就可以通过。

也可以使用 ngx\_http\_core\_module 提供的变量设置新的 location。例如:

```
proxy_redirect http://localhost:8000/ http://$host:$server_port/;
```

也可以省略 replacement 中的主机名, 使用虚拟主机名称。例如:

```
proxy_redirect http://localhost:8000/two/ /one/;
```

使用 off 时, 则不进行转换。

使用 default 参数时, 会按照 proxy\_pass 配置项和所属的 location 配置项重组发送给客户端的 location 头部。例如, 下面两种配置效果一样:

```
location /one/{
    proxy_pass http://upstream:port/two/;
    proxy_redirect default;
}

location /one/{
    proxy_pass http://upstream:port/two/;
    proxy_pass http://upstream:port/two/ /one/;
}
```

### 63. 更换上游服务器

语法:

```
proxy_next_upstream[error|timeout|invalid_header|http_500|http_502|
http_503|http_504|http|404|off];
```

默认:

```
proxy_next_upstream error timeout;
```

配置块: http、server、location。

说明: 表示向上台上游服务器转发请求错误时, 继续换一台上游服务器处理这个请求。因为上游服务器一旦开始发送应答, 反向代理服务器会立刻把应答包转发给客户端, 所以, 一旦 Nginx 开始向客户端发送响应包, 之后的过程中若出现错误不允许换下一台上游服务器继续处理。proxy\_next\_upstream 的参数用来说明在哪些情况下会继续选择下一台上游服

务器转发请求。

可以使用的值和参数如下：

- error：上游服务器出现错误。
- timeout：发送请求或读取响应发生超时。
- invalid\_header：上游服务器发送的响应不合法。
- http\_500：上游服务器返回状态码是 500。
- http\_502：上游服务器返回状态码是 502。
- http\_503：上游服务器返回状态码是 503。
- http\_504：上游服务器返回状态码是 504。
- http\_404：上游服务器返回状态码是 404。
- off：关闭这个功能。

9.3.4 ngx\_http\_core\_module 变量

ngx\_http\_core\_module 模块提供了很多变量，可以在配置日志格式或 URI 中使用。表 9-1 列出了 ngx\_http\_core\_module 变量。

表 9-1 ngx\_http\_core\_module 变量

参数	意义
\$arg_PARAMETER	HTTP 请求中某个参数的值，如 /index.html?id=001，可以使用 \$arg_id 获取 001 这个值
\$args	HTTP 请求中的完整参数字符串，上项中的 \$args 是 “id=001”
\$binary_remote_addr	二进制的客户端地址
\$body_bytes_sent	表示发送给客户端应答包体字节数
\$content_length	客户端请求头中的 Content-Length 头域值
\$content_type	请求头中的 Content-Type 字段
\$cookie_COOKIE	请求头中的 cookie 字段
\$document_root	当前请求使用的 root 配置选项（root 配置指令的值）
\$uri	表示当前请求的 URI，不含参数部分
\$document_uri	与 \$uri 含义相同
\$request_uri	表示请求的原始 URI，包含完整参数。本变量在重定向后也不变
\$host	请求中的 host 字段，只返回 IP，全部小写。如果 host 字段不存在，则使用实际处理的 server（虚拟主机）替代。\$http_HEADER 取出的 http_host 是原始的 host 值，不更改
\$http_HEADER	取出相应的请求头，如 host 为 \$http_host，头域名全部小写
\$hostname	Nginx 所在主机名
\$sent_http_HEADER	HTTP 应答头中相应的头域值，HEADER 全小写，如 \$sent_http_host 是应答的 host 头
\$is_args	表示请求中的 URI 是否带参数，如是带参数值为 “?”（一个问号），不带返回空字符串
\$limit_rate	当前连接限速多少，0 表示无限速
\$nginx_version	当前 Nginx 版本号
\$query_string	请求 URI 中的参数，与 \$args 相同，但是 \$query_string 是只读的，不会改变
\$remote_addr	客户端地址

(续)

参数	意义
\$remote_port	客户端地址
\$remote_user	使用 Auth Basic Module 时定义的用户名
\$request_filename	请求中的 URI 经过 root 或 alias 转换后的文件路径
\$request_body	请求的包体, 只在 proxy_pass 或 fastcgi_pass 中有意义
\$request_body_file	请求包体临时存储的文件名
\$request_completion	请求全部处理完成, 值为“ok”, 否则都是空字符串
\$request_method	HTTP 请求的方法, 如 GET、PUT
\$schema	HTTP scheme, 如请求 https://nginx.com/, 值是 https
\$server_addr	服务器地址
\$server_name	服务器名字
\$server_port	服务器端口
\$server_protocol	服务器发送响应时的协议, 如 HTTP/1.1 或 HTTP/1.0

### 9.3.5 stream

从 1.9.0 版本开始, Nginx 支持 stream 模块, 跟 main、HTTP 和 event 一样, Stream 属于第一层配置块, 实现 TCP 功能。

本模块默认是不在版本里的, 需要通过 --with-stream 配置参数使能。

下面是官方的一个 stream 配置示例:

```
worker_processes auto;
error_log /var/log/nginx/error.log info;
events {
    worker_connections 1024;
}

stream {
    upstream backend {
        hash $remote_addr consistent;
        server backend1.example.com:12345 weight=5;
        server 127.0.0.1:12345 max_fails=3 fail_timeout=30s;
        server unix:/tmp/backend3;
    }

    server {
        listen 12345;
        proxy_connect_timeout 1s;
        proxy_timeout 3s;
        proxy_pass backend;
    }

    server {
        listen [::1]:12345;
        proxy_pass unix:/tmp/stream.socket;
    }
}
```

示例中定义了一个 TCP 的 stream 块，从 1.9.0 开始，Nginx 支持 TCP 纯代理服务。可以利用 Nginx 的负载均衡功能，或 Nginx 强大的网络接入能力实现会话管理和接入，再把具体的业务分发到后面使用其他语言实现的服务器上。

示例中定义了一个由 3 台服务器组成的负载均衡组 (upstream) backend，负载均衡的策略是根据客户端 IP 地址进行 hash，保证相同 IP 过来的总是被调度到同一个服务器上处理，优点是会话可以持续。3 台服务器分配了不同的权重，最后一台服务器是 UNIX 域套接字，是内部通信的服务。另外定义了两个监听 (server)：第一个是 IPv4 的监听，在 12345 端口上；第二个是 IPv6 的监听，也在 12345 端口上。IPv4 的请求交由负载均衡组 backend 处理，IPv5 请求的直接交给 UNIX 域套接字监听的服务处理，不经过负载均衡，这是为了演示两种应用。

下面再给出一个 TCP 服务的配置示例：

```
user root;
worker_processes 4;
worker_rlimit_nofile 100000;

error_log logs/error.log;
pid logs/nginx.pid;

events{
    use epoll;
    worker_connections 10000;
}

stream{
    tcp_nodelay on;

    lua_shared_dict gvar 50m;
    lua_shared_dict gmsg 50m;
    lua_shared_dict gdata 300m;
    lua_shared_dict gsess 30m;
    lua_package_path "/usr/local/lib/lua/5.1/?..lua;;";
    init_worker_by_lua_file init.lua;

    server{
        listen 127.0.0.1:8081;
        listen 10.47.105.112:8081;
        listen 120.27.148.219:8081;

        lua_socket_log_errors off;
        content_by_lua_file mmp.lua;
    }
}

http{
    include mime.types;
    default_type text/html;
```



```

access_log off;
server_tokens    off;

sendfile         on;
tcp_nopush       on;
tcp_nodelay      on;
open_file_cache  max=10240 inactive=60s;
open_file_cache_valid 80s;
open_file_cache_min_uses 1;
lua_shared_dict  gkey 50m;

keepalive_timeout 0;
chunked_transfer_encoding off;
lua_package_path  "/usr/local/lib/lua/5.1/??.lua;;";
.....
}

```

本例中，在 8081 端口监听了一个 TCP 服务，工作进程初始化时使用 init 脚本初始化了预定义的共享字典，TCP 内容进来时使用 mmp.lua 脚本处理业务逻辑。而 HTTP 部分定义了一些内部的 location 为 TCP 服务使用（参考前面的 location 例子，此处略）。

## 1. stream 配置指令

### (1) 监听

语法：

```
listen address:port [ssl] [udp] [proxy_protocol] [backlog=number] [bind]
[ipv6only=on|off] [reuseport] [so_keepalive=on|off|[keepidle]:[keepintvl]:[keepc
nt]];
```

默认：无。

配置块：server。

说明：设置监听的套接字 IP 地址和端口。也可以只指定端口。地址可以是主机名，例如：

```

listen 127.0.0.1:12345;
listen *:12345;
listen 12345;      # same as *:12345
listen localhost:12345;

```

支持 IPv6 地址，地址用 [] 括起来：

```

listen [::1]:12345;
listen [::]:12345;

```

UNIX 域套接字使用 “unix:” 前缀：

```
listen unix:/var/run/nginx.sock;
```

ssl 参数用于声明所有连接使用 SSL 模式。

udp 参数声明这是一个 UDP 监听。

proxy\_protocol 参数指定所有连接使用 Proxy 协议。

listen 指令可以有几个参数：

- backlog=number：设置 listen() 调用时未决状态连接队列大小。默认地，该参数在 Free 子 BSD 系统上是 -1，在其他系统是 511。
- bind：如果这个 listen 使用 bind 参数，表示实际绑定在这个 ip\*port 对上。同样，如果哪一行 ipv6only 或者 so\_keepalive 参数在一个 address:port 指定，则与 bind 参数一样，会执行绑定操作。
- ipv6only=on|off：这个参数表明是否在 [::] 上是只监听 IPv6 连接还是 IPv6 和 IPv4 同时都接收。这个参数默认打开，并且只能在启动时打开一次。
- reuseport：这个参数使每一个工作进程都在 IP 和端口上监听一次（使用 so\_reuseport 套接字选项），允许内核在工作进程间分配任务。目前仅 Linux 3.9+ 和 DragonFly 支持该配置项。
- so\_keepalive=on|off[keepidle]:[keepintvl]:[keepcnt]：这个指令为监听的套接字配置 TCP 保活机制。操作系统的设备可能会忽略这个参数。如果该参数设置为 on，则套接字 so\_keepalive 参数会被启用。

## （2）预读缓冲区

语法：

```
preread_buffer_size size;
```

默认：

```
preread_buffer_size 16k;
```

配置块：stream、server。

说明：指定预读缓冲区尺寸。

## （3）预计超时

语法：

```
preread_timeout timeout;
```

默认：

```
preread_timeout 30s;
```

配置块：stream、server。

说明：指定预读阶段超时值。

## （4）Proxy 协议

语法：

```
proxy_protocol_timeout timeout
```

默认：

```
proxy_protocol_timeout 30s;
```

配置块: stream、server。

说明: 指定 Proxy 协议头读取超时值。如果这期间没有头数据传输, 连接将被关闭。

### (5) 域名解析服务

语法:

```
resolver address ... [valid=time] [ipv6=on|off]
```

默认: 无。

配置块: stream、server。

说明: 配置域名服务器, 用于解析地址中的 upstream 服务。例如:

```
resolver 127.0.0.1 [::1]:5353;
```

将一个地址指定一个域名, 可选配端口。如果未指定端口, 使用 53 号端口。默认地, Nginx 将在 IPv4 和 IPv6 中查找解析。如果没有 IPv6 地址, 可以设置 `ipv6=off`。默认地, Nginx 使用应答的 TTL 值缓存结果, `valid` 参数可以改写这个值:

```
resolver 127.0.0.1 [::1]:5353 valid=30s;
```

### (6) 解析超时值

语法:

```
resolver_timeout time;
```

默认:

```
resolver_timeout 30s;
```

配置块: stream、server。

说明: 设置域名解析超时值。

### (7) 服务

语法:

```
server{...}
```

默认: 无。

配置块: stream。

说明: 配置一个 server 块, server 中可用的指令均可用在 stream 下的 server 中。

### (8) tcp\_nodelay

语法:

```
tcp_nodelay on|off
```

默认:

```
tcp_nodelay on;
```

配置块：stream、server。

说明：使能 tcp\_nodelay 选项。

### (9) 变量 hash 表桶尺寸

语法：

```
variables_hash_bucket_size size;
```

默认：

```
variables_hash_bucket_size 64;
```

配置块：stream。

说明：设置变量 hash 表桶尺寸。

### (10) 变量 hash 表最大尺寸

语法：

```
variables_hash_max_size size;
```

默认：

```
variables_hash_max_size 1024;
```

配置块：stream。

说明：设置变量哈希表最大尺寸。

## 2. stream 内建变量

ngx\_stream\_core\_module 模块从 1.11.2 版本开始支持下面的变量：

- \$binary\_remote\_addr：二进制的客户端地址，IPv4 总是 4 字节，IPv6 总是 16 字节。
- \$bytes\_received：从客户端收到的字节数（1.11.4）。
- \$bytes\_sent：发送给客户端的字节数。
- \$connection：连接序列号。
- \$hostname：主机名。
- \$msec：当前时间，精度是毫秒。
- \$nginx\_version：Nginx 版本。
- \$pid：工作进程 PID。
- \$protocol：和客户端通信的协议——TCP 或 UDP（1.11.4）。
- \$proxy\_protocol\_addr：Proxy 协议头中的客户端地址，也可能是一个空字符串。
- \$proxy\_protocol\_port：Proxy 协议头中的客户端端口，或者用一个空字符串代替（1.11.4）。
- \$remote\_addr：客户端地址。
- \$remote\_port：客户端端口。
- \$server\_addr：服务器地址。

- \$server\_port: 服务器端口。
- \$session\_time: 会话时间, 精度是毫秒。
- \$status: 会话状态 (1.11.4), 可以是下列值。
  - ✓ 200: 成功。
  - ✓ 400: 客户数据不能解析。
  - ✓ 403: 禁止访问。
  - ✓ 500: 服务器内部错误。
  - ✓ 502: 错误路由。
  - ✓ 503: 服务不可达。
- \$time\_iso8601: ISO 8601 格式的本地时间。
- \$time\_local: 通常 Log 格式的本地时间。

## 9.4 中文版 nginx.conf

下面是一份 cnblog 上的中文版 nginx.conf 示例, 其中大部分配置指令本文都介绍过, 请参照学习。

```
# $ 开头是变量
# 定义 Nginx 运行的用户和用户组
user work work;
# Nginx 进程数, 建议设置为 CPU 总核心数
worker_processes auto;
# 指当一个 Nginx 进程打开的最多文件描述符数目
worker_rlimit_nofile 204800;
# 全局错误日志定义类型, [ debug | info | notice | warn | error | crit ]
error_log /opt/log/nginx/error.log error;
# 工作模式及连接数上限
events {
    # 参考事件模型, use [ kqueue | rtsig | epoll | /dev/poll | select | poll ];
    # epoll 模型是 Linux 2.6 以上版本内核中的高性能网络 I/O 模型, 如果运行在 FreeBSD 上面,
    # 就用 kqueue 模型
    use epoll;
    # 单个后台 worker process 进程的最大并发连接数
    worker_connections 102400;
}
# 设定 http 服务器, 利用它的反向代理功能提供负载均衡支持
http {
    # 文件扩展名与文件类型映射表
    include mime.types;
    # 默认文件类型
    default_type application/octet-stream;
    # 默认编码
    charset utf-8;
    # 设定日志格式
    log_format main '$idXXXX\t$remote_addr\t$remote_user\t$time_local\t$host\t$request\t';
```

```

# '$status\t$body_bytes_sent\t$http_referer\t'
# '$http_user_agent\t$http_x_forwarded_for\t$request_time\t$upstream_
response_time\t$userid';
log_format main "$cookie_idXXXX\t$remote_addr\t$remote_user\t[$time_
local]\t$request_method\t$host\t$request_uri\t"
"$request_time\t$status\t$body_bytes_sent\t'$http_
referer'\t"
"$http_user_agent'\t'$http_x_forwarded_for'\t$upstream_
addr\t$upstream_response_time\t$upstream_status\t";

```

# 不可见字符分隔日志格式

```
#include other_log_format.conf;
```

# 实时日志收集 JSON 格式日志

```
#include json_log_format.conf;
```

# 日志流格式

```

log_format stream_log "$cookie_idXXXX\t$remote_addr\t$remote_user\
t[$time_local]\t$request_method\t$host\t$request_uri\t"
"$request_time\t$status\t$body_bytes_sent\t'$http_referer'\t" "'$http_user_
agent'\t'$http_x_forwarded_for'\t$upstream_addr\t$upstream_response_time\t3";

```

# 成功日志

```
access_log /opt/log/nginx/access.log main;
```

```
#access_log syslog:local6:notice:log1.op.XXXXdns.org:514:nginx-main-log
main;
```

# 指定 Nginx 是否调用 sendfile 函数 (zero copy 方式) 来输出文件, 对于普通应用,

# 必须设为 on, 如果用来进行下载等应用磁盘 I/O 重负载应用, 可设置为 off, 以平衡磁盘与网  
络 I/O 处理速度, 降低系统的 uptime

```
sendfile on;
```

# 长连接超时时间, 单位是秒

```
keepalive_timeout 60;
```

# 服务器名称 hash 表的最大值 (默认 512)[hash%size]

```
server_names_hash_max_size 1024;
```

# 服务器名字的哈希表大小

```
server_names_hash_bucket_size 256;
```

# 客户请求头缓冲大小

```
client_header_buffer_size 4k;
```

# 如果 header 过大, 它会使用 large\_client\_header\_buffers 来读取

```
large_client_header_buffers 4 256k;
```

```
client_header_timeout 1m;
```

```
client_body_timeout 1m;
```

```
send_timeout 1m;
```

# 防止网络阻塞

```
tcp_nopush on;
```

```
tcp_nodelay on;
```

# 允许客户端请求的最大单文件字节数

```
client_max_body_size 50m;
```

# 缓冲区代理缓冲用户端请求的最大字节数

```
client_body_buffer_size 50m;
```

#Nginx 跟后端服务器连接超时时间 (代理连接超时)

```
proxy_connect_timeout 5;
```

```

# 后端服务器数据回传时间 (代理发送超时)
proxy_send_timeout 15;

# 连接成功后, 后端服务器响应时间 (代理接收超时)
proxy_read_timeout 15;

# 设置代理服务器 (Nginx) 保存用户头信息的缓冲区大小
proxy_buffer_size 4k;

# proxy_buffers 缓冲区, 网页平均在 32KB 以下的话, 这样设置
proxy_buffers 8 32k;

# 高负荷下缓冲大小 (proxy_buffers*2)
proxy_busy_buffers_size 64k;

# 设定缓存文件夹大小, 大于这个值, 将从 upstream 服务器传
proxy_temp_file_write_size 64k;
proxy_intercept_errors on;
# 客户端放弃请求, Nginx 也放弃对后端的请求
# proxy_ignore_client_abort on;

# 代理缓存头信息最大长度 [ 设置头部哈希表的最大值, 不能小于后端服务器设置的头部总数 ]
proxy_headers_hash_max_size 512;
# 设置头部 hash 表大小 (默认 64) [ 这将限制头部字段名称的长度大小, 如果使用超过 64 个字符的头
部名可以加大这个值。 ]
proxy_headers_hash_bucket_size 256;

# 变量 hash 表的最大值 (默认值)
variables_hash_max_size 512;
# 为变量 hash 表设置关键字栏的大小 (默认 64)
variables_hash_bucket_size 128;

# 开启 gzip 压缩输出
gzip on;
# 最小压缩文件大小
gzip_min_length 1k;
# 压缩缓冲区
gzip_buffers 4 16k;
# 压缩等级
gzip_comp_level 9;
# 压缩版本 (默认 1.1, 前端如果是 squid2.5, 则使用 1.0)
gzip_http_version 1.0;
# 压缩类型, 默认就已经包含 text/xml
gzip_types text/plain application/x-javascript application/json application/
javascript text/css application/xml text/javascript image/gif image/png;
gzip_vary on;

# map 模块的使用
map_hash_max_size 102400;
map_hash_bucket_size 256;

# Tengine Config
# concat on;
# trim on;
# trim_css off;

```

```

#trim_js off;
server_tokens off;
#footer "<!-- $remote_addr $server_addr $upstream_addr -->";

#rewrite_log on;
fastcgi_intercept_errors on;
#include other config file
include ../conf.d/*.conf;
# 包含一些特殊站点的配置文件, 此目录下文件暂时不包含在 git 自动管理过程中
include ../special/*.conf;

# 屏蔽不加主机域名的默认请求
#server {
#    listen *:80 default;
#    server_name _ "";
#    return 444;
#}

#ceshi by dongange 2016-01-07

#Nginx 状态监测模块配置
req_status_zone server "$server_name,$server_addr:$server_port" 10M;
req_status server;
server {
    listen 127.0.0.1:80;
    server_name 127.0.0.1;
    access_log /opt/log/nginx/nginx_status/status_access.log main;
    location /status {
        req_status_show;
        access_log /opt/log/nginx/nginx_status/status_access.log main;
        allow 127.0.0.1;
        deny all;
    }
    location /stub_status {
        stub_status on;
        access_log /opt/log/nginx/nginx_status_stub/status_stub_access.
log main;

        allow 127.0.0.1;
        deny all;
    }
    location /check_status {
        check_status;
        access_log /opt/log/nginx/nginx_status_check/status_access_
check.log main;

        allow 127.0.0.1;
        deny all;
    }
}
}

```

## 9.5 小结

本章介绍了 Nginx 中的默认配置文件, Nginx 是通过 nginx.conf 进行配置的。同时本章



给出一个包含有 Redis 访问、MySQL 访问的 nginx.conf 配置实例，方便读者对配置文件有一个总体的认识。

本章还对 Nginx 的顶层配置块和全局配置块（Nginx 共有 main 全局配置、events 配置块、http 配置块、stream 配置块 5 个顶层配置块），以及各配置块所属的配置指令进行了详细介绍。

Nginx 的配置内容是由模块决定的，本章介绍的配置是系统核心模块配置指令。系统中使用了不同的模块后，根据使用到的模块，还会有新的配置项。例如，后续我们使用 ngx\_lua 模块进行 Lua 开发时，还需要在本章介绍的配置基础上，增加 ngx\_lua 模块需要用到的配置指令。

## Nginx 下 Lua 实现机制

Nginx 中的大部分功能是通过模块提供的，这种方式使得 Nginx 开发和扩展很方便，模块可以依次串起来形成一个过滤器。一个模块的失效不会影响其他部分，是 Nginx 扩展性和可靠性的一个保证。系统提供了如 http 模块、mail 模块、event 等模块。根据业务需要，通过配置或编译将不同的模块组合起来，形成自己业务特有的 Web 服务器，实现一些特定的功能。Nginx 使用比较多的场合是反向代理，使用 Nginx 在前端做登录校验、JS 合并、数据库访问、访问鉴权等，具体业务由反向代理实现，发挥 Nginx 业务开发灵活、有大并发接入和会话保持能力的优点。

以往 Nginx 模块是通过 C 或 C++ 开发的，需要符合 Nginx 的开发规则和数据结构。使用 C 开发 Nginx 模块必须要熟悉 Nginx 源码，而且 C 下面模块的调试非常麻烦。在以 Nginx 作为核心开发业务系统时，大量的数据库、缓存等访问会使得开发周期非常漫长。chaoslawful 和 agentzh 将 Lua 解释器集成进 Nginx 中，开发了 ngx\_lua 模块，使开发者可以使用 Lua 脚本语言实现业务。Lua 是高效、紧凑的脚本语言，为了嵌入式而开发的快速脚本语言，内建协程，非常便于业务开发，所以使用 Lua 脚本语言快速地开发高并发业务系统可以大大降低开发工作量，使业务开发更灵活，更快速。ngx\_lua 也使得 Nginx 得到了更广泛的应用，很多工程师使用 Lua+Nginx 开发了许多富有创意的应用。

### 10.1 ngx\_lua 原理

ngx\_lua 将 Lua 集成进 Nginx。Lua 内建协程，使用协程可以很好地将异步回调转换成顺序调用的形式，和 Nginx 的全异步模式匹配起来：协程调用异步 API，然后协程挂起，

在异步回调事件到来时，再将协程唤醒，继续执行。这样既可以实现全异步的 Nginx 机制，不会影响 Nginx 的高并发处理性能，又使开发者以同步的方式编写异步程序，使代码复杂性大为降低。ngx\_lua 中 Lua 的 I/O 操作都委托给 Nginx 事件模型，从而实现完全的非阻塞调用。协程在挂起时自动保存上下文，工作进程上的 VM 可以处理下一个协程的任务。底层事件模型完成任务时，根据回调，对应的例程会被恢复上下文，从而继续执行用户操作，这个过程不需要对数据进行同步和匹配，用起来与同步操作无异。

Nginx 采用管理 / 工作进程 (master/worker) 的主从进程机制，工作进程都由管理进程管理，同时配置文件、命令、任务分派由管理进程实现，从而使 Nginx 可以动态重新载入配置文件、动态升级、动态部署。而多工作进程的机制，可以充分利用多处理器架构的性能，又不会因为过多的线程增加调试的开销。而工作进程时刻处于管理进程的监管之下，任何崩溃可以由管理进程马上重启一个工作进程，实现高可靠性。

图 10-1 描述了 Nginx 管理 / 工作进程模式。

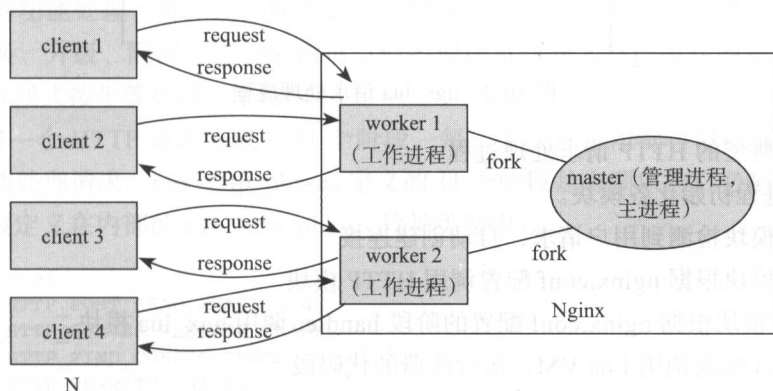


图 10-1 Nginx 管理 / 工作进程模式

ngx\_lua 在 Nginx 的管理 / 工作进程机制基础上加入了 Lua 解释器或虚拟机。ngx\_lua 在每一个 Nginx 工作进程上执行一个 Lua 解释器或 LuaJIT 实例 (VM 虚拟机实例)，这个工作进程上处理的所有请求共享这个实例。每个请求的上下文都被 Lua 的协程分割，保证每个请求是独立的。ngx\_lua 采用“一个请求一个协程”的处理模型，对于每个用户请求，ngx\_lua 都会创建一个协程用于执行用户代码以处理请求，请求处理完毕，协程会被销毁。每个协程都有一个独立的全局变量 (变量空间)，保存从全局共享的数据。所以，用户在自己的代码里操作了全局空间的变量不会影响其他的请求处理协程，因为数据被隔离了，数据用完了会被释放，用户代码运行在一个沙箱 (sandbox) 内，沙箱的生存周期与请求的生命周期相同，请求时被分配并创建协程，处理完毕刚被释放。

协程是轻量级线程，所以占用极少内存，通常每个请求 ngx\_lua 只会占用 2KB 左右内存。协程不能同时运行。

图 10-2 描述了一个简单的 ngx\_lua 请求处理流程。

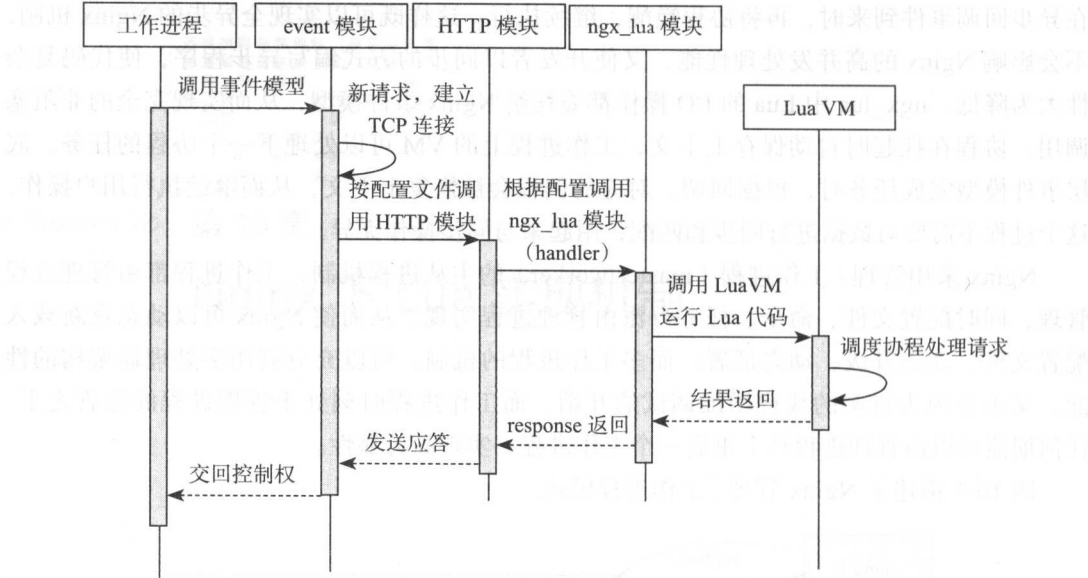


图 10-2 ngx\_lua 请求处理流程

这是一个典型的 HTTP 请求处理过程：

- 1) 工作进程初始化各模块。
- 2) event 模块检测到用户请求，自动创建连接。
- 3) event 模块根据 nginx.conf 配置调用 HTTP 模块。
- 4) HTTP 模块根据 nginx.conf 配置的阶段 handler 调用 ngx\_lua 模块。
- 5) ngx\_lua 模块调用 Lua VM，运行注册的代码段。
- 6) Lua VM 调用协程处理代码。
- 7) VM 向 ngx\_lua 返回结果数据。
- 8) ngx\_lua 向 HTTP 模块返回 HTTP 应答。
- 9) HTTP 调用 event 模块发送回答。
- 10) event 模块向工作进程交还控制权。

现在 Nginx 也支持 stream 模块、websocket 协议等，处理过程有所不同，但总体机制是一样的。

ngx\_lua 模块的主要机制总结如下：

- 1) 每个工作进程创建一个 Lua 虚拟机 (VM)，工作进程内所有协程使用相同 VM。
- 2) 将 Nginx I/O 操作封装成 Lua 使用的 API 载入虚拟机，供 Lua 代码使用。
- 3) 每个外部请求都由一个 Lua 协程处理，协程之间数据隔离。
- 4) Lua 代码调用 I/O 等异步接口时，当前协程会被挂起，上下文数据自动保存，不会阻塞工作进程。

5) I/O 等异步操作完成时还原线程上下文，继续代码执行。

ngx\_lua 模块提供了新的配置项以支持 Lua 编程，同时 ngx\_lua 支持 Nginx 的主要阶段，通过类似于 content\_by\_lua\*、access\_by\_lua\*、init\_by\_lua\*、init\_worker\_by\_lua\* 等配置项将 Lua 代码挂载入 Nginx 的事件处理机制中。Nginx 启动时虚拟机 VM 被创建，同时配置项被装载，当 Nginx 框架处理请求事件，对应的阶段发生调用时，注册的 Lua 代码实现的阶段 handler 接管输入管道和输出管道，在 VM 处理完后，将数据返回给调用者，实现的 Lua 介入 Nginx 处理。

## 10.2 HTTP 请求的处理阶段

基于 Nginx 使用的多模块设计思想，Nginx 将 HTTP 请求的处理过程划分为很多个阶段。这样可以使一个 HTTP 请求的处理过程由很多模块参与处理，每个模块只专注于一个独立而简单的功能处理，可以使性能更好、更稳定，同时拥有更好的扩展性。这些模块可以灵活地更换、升级、隔离。划分阶段可以使各模块可以介入自己关注的处理阶段中，使各模块以流水线式按步骤处理一个请求。

Nginx 将一个 HTTP 请求分成了 11 个阶段，每一个处理阶段都可以由任意多个 HTTP 模块流水式地处理请求。ngx\_http\_phases 定义的 11 个阶段是有顺序的，必须按照定义的顺序执行。阶段定义在内部的 ngx\_http\_phases 枚举类型中：

```
typedef enum{
    NGX_HTTP_POST_READ_PHASE = 0,
    NGX_HTTP_SERVER_REWRITE_PHASE,
    NGX_HTTP_FIND_CONFIG_PHASE,
    NGX_HTTP_REWRITE_PHASE,
    NGX_HTTP_POST_REWRITE_PHASE,
    NGX_HTTP_PREACCESS_PHASE,
    NGX_HTTP_ACCESS_PHASE,
    NGX_HTTP_POST_ACCESS_PHASE,
    NGX_HTTP_TRY_FILES_PHASE,
    NGX_HTTP_CONTENT_PHASE,
    NGX_HTTP_LOG_PHASE
}ngx_http_phases;
```

这 11 个阶段的意义分别如下：

- 1) NGX\_HTTP\_POST\_READ\_PHASE：接收到完整的 HTTP 头部后的处理阶段。
- 2) NGX\_HTTP\_SERVER\_REWRITE\_PHASE：在将请求的 URI 与 location 表达式匹配前，修改请求的 URI（重定向），是一个独立的 HTTP 阶段。
- 3) NGX\_HTTP\_FIND\_CONFIG\_PHASE：根据请求的 URI 寻找匹配的 location 表达式，这个阶段只能由 ngx\_http\_core\_module 模块实现，不建议其他 HTTP 模块重新定义这一阶段行为。
- 4) NGX\_HTTP\_REWRITE\_PHASE：在 NGX\_HTTP\_FIND\_CONFIG\_PHASE 阶段寻

找到匹配的 location 之后再修改请求的 URI。

5) `NGX_HTTP_POST_REWRITE_PHASE`：重写 URL 后，防止错误的 `nginx.conf` 配置导致死循环（递归地修改 URI），这一阶段仅由 `ngx_http_core_module` 模块处理。目前控制死循环的方式是限制 10 次重定向，超过 10 次，本阶段向会用户返回 500 错误。

6) `NGX_HTTP_PREACCESS_PHASE`：这是下一个阶段（`NGX_HTTP_ACCESS_PHASE`）决定请求的访问权限前，HTTP 模块可以介入的处理阶段。

7) `NGX_HTTP_ACCESS_PHASE`：用于 HTTP 模块判断是否允许这个请求访问 Nginx 服务器。

8) `NGX_HTTP_POST_ACCESS_PHASE`：当上一个阶段的 handler 函数返回不允许访问的错误码（`NGX_HTTP_FORBIDDEN`、`NGX_HTTP_UNAUTHORIZED`），这个阶段负责向用户发送拒绝服务的错误响应。用于上一阶段的收尾处理。

9) `NGX_HTTP_TRY_FILES_PHASE`：这个阶段是为了 `try_files` 设置的，当 HTTP 请求访问静态文件资源时，`try_files` 配置项可以使这个请求顺序地访问多个静态文件资源，如果某一次访问失败，则继续访问 `try_files` 中指定的下一个静态资源。

10) `NGX_HTTP_CONTENT_PHASE`：用于处理 HTTP 请求内容的阶段，这是大部分 HTTP 模块最常用的介入阶段。

11) `NGX_HTTP_LOG_PHASE`：日志写入阶段。`ngx_http_log_module` 实际上也是在这里注册了一个 handler 实现的日志写入工作。

这些阶段有些是必备的，有些是可选的，也有一些是可以多次进入的：

`NGX_HTTP_POST_READ_PHASE`、`NGX_HTTP_SERVER_REWRITE_PHASE`、`NGX_HTTP_REWRITE_PHASE`、`NGX_HTTP_PREACCESS_PHASE`、`NGX_HTTP_ACCESS_PHASE`、`NGX_HTTP_CONTENT_PHASE`、`NGX_HTTP_LOG_PHASE` 允许用户介入操作。其他的 4 个阶段是由 HTTP 框架实现，不允许用户进入。

### 10.3 ngx\_lua 的处理阶段

`ngx_lua` 在 HTTP 处理阶段基础上，分别在 Rewrite/Access 阶段、Content 阶段、Log 阶段注册了自己的 handler，加上系统初始阶段 master 的两个阶段，共 11 个阶段为 Lua 脚本提供处理介入的能力。

`ngx_lua` 的 11 个阶段是建立在 HTTP 阶段之上的，所以并不完全等同于 HTTP 的 11 个阶段。

图 10-3 描述了 Lua 的 11 个阶段在 Nginx 的 4 个主要阶段分布情况。

参照 HTTP 的核心阶段，加上 master 进程的几个重要阶段，就构成了 `ngx_lua` 可以介入的阶段，另外指令可以在 `http`、`server`、`server if`、`location`、`location if` 几个范围进行配置。

表 10-1 描述了 Lua 可以使用的主要阶段。

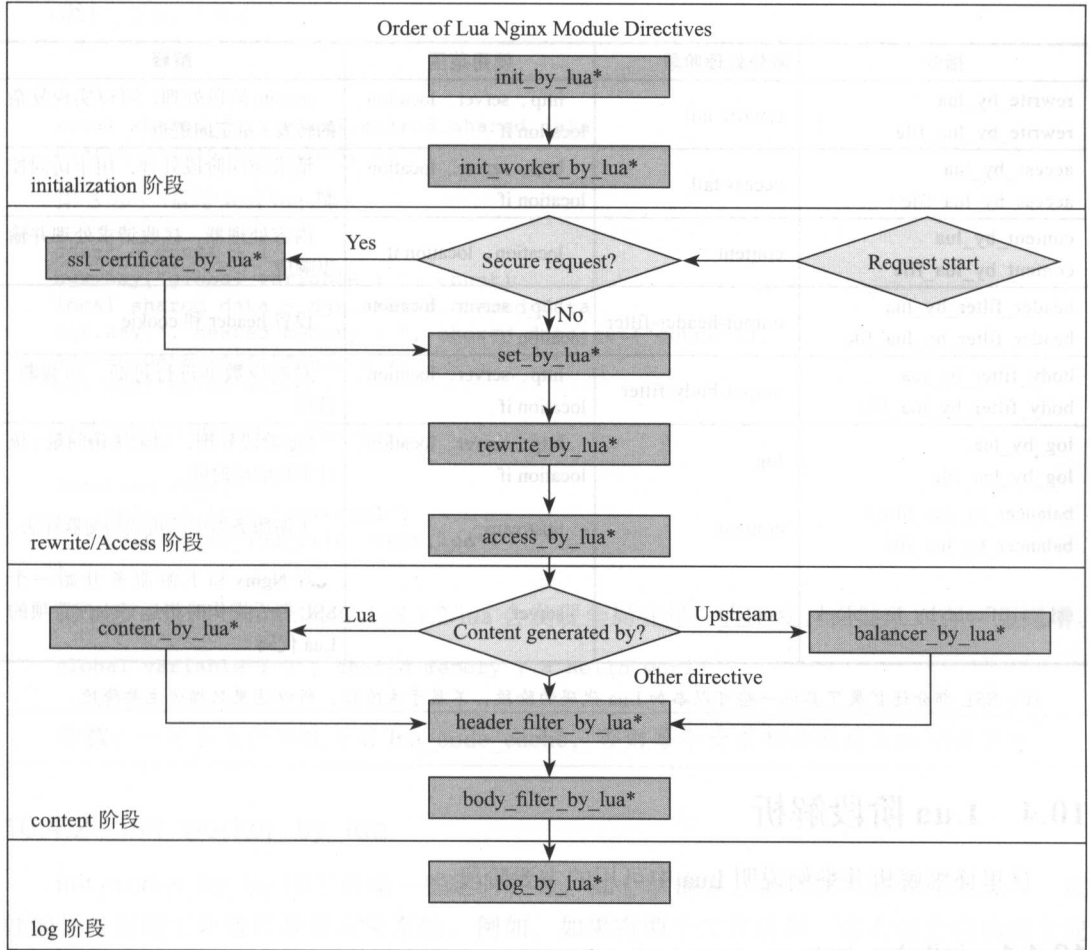


图 10-3 ngx\_lua 的 11 个用户可介入阶段

表 10-1 Lua 可使用的主要阶段

指令	所处处理阶段	使用范围	解释
init_by_lua init_by_lua_file	loading-config	http	Nginx 管理进程加载配置时执行；通常用于初始化全局配置/预加载 Lua 模块
init_worker_by_lua init_worker_by_lua_file	starting-worker	http	每个 Nginx 工作进程启动时调用的计时器，如果管理进程不允许则只会在 init_by_lua 之后调用；通常用于定时拉取配置/数据，或者进行后端服务的健康检查
set_by_lua set_by_lua_file	rewrite	server、server if、location、location if	设置 Nginx 变量，可以实现复杂的赋值逻辑；此处是阻塞的，Lua 代码要做到非常快

(续)

指令	所处处理阶段	使用范围	解释
rewrite_by_lua rewrite_by_lua_file	rewrite tail	http、server、location、 location if	rewrite 阶段处理，可以实现复杂的转发 / 重定向逻辑
access_by_lua access_by_lua_file	access tail	http、server、location、 location if	请求访问阶段处理，用于访问控制
content_by_lua content_by_lua_file	content	location、location if	内容处理器，接收请求处理并输出响应
header_filter_by_lua header_filter_by_lua_file	output-header-filter	http、server、location、 location if	设置 header 和 cookie
body_filter_by_lua body_filter_by_lua_file	output-body-filter	http、server、location、 location if	对响应数据进行过滤，如截断、替换
log_by_lua log_by_lua_file	log	http、server、location、 location if	log 阶段处理，如记录访问量 / 统计平均响应时间。
balancer_by_lua_block balancer_by_lua_file	content	upstream	上游服务器中的负载均衡器算法。
ssl_certificate_by_lua_block	content	server	在 Nginx 和下游服务开始一个 SSL 握手操作时将运行本配置项的 Lua 代码

注：SSL 部分还扩展了其他一些可以添加 Lua 代码的阶段，不属于主阶段，所以这里只描述主要阶段。

## 10.4 Lua 阶段解析

这里详细解析并举例说明 Lua 中可用的主要阶段。

### 10.4.1 init\_by\_lua

init\_by\_lua 在每次 Nginx 重新加载配置时执行，可以用来完成一些耗时模块的加载，或者初始化一些全局配置。在管理进程创建工作进程时，此指令中加载的全局变量会进行 Copy-OnWrite，即会复制所有全局变量到工作进程中。

第 1 步：在 nginx.conf 配置文件中的 http 部分添加如下代码。

```
# 共享全局变量，在所有工作进程间共享
lua_shared_dict shared_data 1m;

init_by_lua_file /usr/example/lua/init.lua;
```

第 2 步：编写 init.lua。

```
-- 初始化耗时的模块
local redis = require 'resty.redis'
local cJSON = require 'cjson'
```



```
-- 全局变量, 不推荐
count = 1
```

```
-- 共享全局内存
local shared_data = ngx.shared.shared_data
shared_data:set("count", 1)
```

第 3 步: 缩写 test.lua。

```
count = count + 1
ngx.say("global variable : ", count)
local shared_data = ngx.shared.shared_data
ngx.say(" ", shared memory : ", shared_data:get("count"))
shared_data:incr("count", 1)
ngx.say("hello world")
```

并在 http 中增加 location /lua:

```
location /lua{
    default_type "text/xml";
    content_by_lua_file "test.lua"
}
```

第 4 步: 访问如 http://192.168.1.2/lua 会发现全局变量一直不变, 而共享内存一直递增。

```
global variable : 2 , shared memory : 8 hello world
```

---

**注意:** 一定在生产环境开启 lua\_code\_cache, 否则每个请求都会创建 Lua VM 实例。

---

## 10.4.2 init\_worker\_by\_lua

init\_worker\_by\_lua 用于启动一些定时任务, 如心跳检查、定时拉取服务器配置等。此处的任务是跟工作进程数量有关系的。例如, 如果有两个工作进程, 那么就会启动两个完全一样的定时任务。

第 1 步: 在 nginx.conf 配置文件中的 http 部分添加如下代码。

```
init_worker_by_lua_file /usr/example/lua/init_worker.lua;
```

第 2 步: 编写 init\_worker.lua 文件, 内容如下。

```
local count = 0
local delayInSeconds = 3
local heartbeatCheck = nil

heartbeatCheck = function(args)
    count = count + 1
    ngx.log(ngx.ERR, "do check", count)

    local ok, err = ngx.timer.at(delayInSeconds, heartbeatCheck)

    if not ok then
        ngx.log(ngx.ERR, "failed to startup heartbeat worker...", err)
```

```
end
end
```

```
heartbeatCheck()
```

ngx.timer.at：延时调用相应的回调方法。可以将延时设置为 0 即得到一个立即执行的任务，任务不会在当前请求中执行，不会阻塞当前请求，而是在一个轻量级线程中执行。

另外，根据实际情况设置如下指令：

```
lua_max_pending_timers 1024; # 最大等待任务数
lua_max_running_timers 256; # 最大同时运行任务数
```

### 10.4.3 set\_by\_lua

set\_by\_lua 用于设置 Nginx 变量，用于处理一些特定的情况，例如，用 set 指令配合 if 指令也很难实现的赋值逻辑，可以在本阶段实现。

第 1 步：在 example.conf 配置文件添加 set\_by\_lua 定义。

```
location /lua_set_1 {
    default_type "text/html";
    set_by_lua_file $num /usr/example/lua/test_set_1.lua;
    echo $num;
}
```

set\_by\_lua\_file 的语法如下：

```
set_by_lua_file $var lua_file arg1 arg2...;
```

该指令在 Lua 代码中可以实现所有复杂的逻辑，但是要执行速度很快，不要阻塞。

第 2 步：定义 test\_set\_1.lua 文件。

编写 test\_set\_q.lua 文件，内容如下：

```
local uri_args = ngx.req.get_uri_args()
local i = uri_args["i"] or 0
local j = uri_args["j"] or 0

return i + j
```

这个文件的功能是得到请求参数进行相加然后返回。

访问如 [http://192.168.1.2/lua\\_set\\_1?i=1&j=10](http://192.168.1.2/lua_set_1?i=1&j=10) 进行测试。如果用纯 set 指令是无法实现的。

再举一个实际例子，我们实际工作时经常涉及网站改版，有时候需要新老并存，或者切一部分流量到新版，下面说明这个过程。

首先在 example.conf 中使用 map 指令映射 host 到指定 Nginx 变量，方便测试。

```
##### 测试时使用的动态请求
map $host $item_dynamic {
    default "0";
    item2014.jd.com "1";
}
```

如绑定 hosts:

```
192.168.1.2 item.jd.com;
192.168.1.2 item2014.jd.com;
```

此时我们想访问 item2014.jd.com 时访问新版, 那么我们可以简单地使用如下指令:

```
if ($item_dynamic = "1") {
    proxy_pass http://new;
}
proxy_pass http://old;
```

假如我们并未完成 8 位编号商品页面的改版(如品类为图书的商品)没有改版完成, 需按照相应规则跳转到老版, 其他页面定向到新版, 虽然使用 if 指令能实现, 但是比较麻烦, 可以这样做:

```
set jump "0";
if($item_dynamic = "1") {
    set $jump "1";
}
if(uri ~ "^/6[0-9]{7}.html") {
    set $jump "${jump}2";
}
# 非强制访问新版, 且访问指定范围的商品
if (jump == "02") {
    proxy_pass http://old;
}
proxy_pass http://new;
```

以上规则是比较简单的, 如果要用更复杂的多重 if/else 或嵌套 if/else 实现, 则可能需要到后端去做, 此时可以借助 Lua:

```
set_by_lua $to_book '
    local ngx_match = ngx.re.match
    local var = ngx.var
    local skuId = var.skuId
    local r = var.item_dynamic ~= "1" and ngx.re.match(skuId, "[0-9]{8}$")
    if r then return "1" else return "0" end;
';
set_by_lua $to_mvd '
    local ngx_match = ngx.re.match
    local var = ngx.var
    local skuId = var.skuId
    local r = var.item_dynamic ~= "1" and ngx.re.match(skuId, "[0-9]{9}$")
    if r then return "1" else return "0" end;
';
# 自营图书
if ($to_book) {
    proxy_pass http://127.0.0.1/old_book/$skuId.html;
}
# 自营音像
if ($to_mvd) {
    proxy_pass http://127.0.0.1/old_mvd/$skuId.html;
}
```

```
# 默认
proxy_pass http://127.0.0.1/proxy/$skuId.html;
```

#### 10.4.4 rewrite\_by\_lua

rewrite\_by\_lua 用于执行内部 URL 重写或者外部重定向，典型的如伪静态化 URL 重写，本阶段在 rewrite 处理阶段的最后默认执行。

示例 1：在 rewrite 阶段实现新老页面跳转。

第 1 步，修改 example.conf 配置文件。

```
location /lua_rewrite_1 {
    default_type "text/html";
    rewrite_by_lua_file /usr/example/lua/test_rewrite_1.lua;
    echo "no rewrite";
}
```

第 2 步，编写 test\_rewrite\_1.lua 文件。

```
if ngx.req.get_uri_args()["jump"] == "1" then
    return ngx.redirect("http://www.jd.com?jump=1", 302)
end
```

当我们请求 `http://192.168.1.2/lua_rewrite_1` 时发现没有跳转，而请求 `http://192.168.1.2/lua_rewrite_1?jump=1` 时发现跳转到京东首页了。此处 301/302 跳转根据自己需求定义。

示例 2：使用 set\_uri\_args 重写请求内部实现重新发起定位。

第 1 步：修改 example.conf 配置文件。

```
location /lua_rewrite_2 {
    default_type "text/html";
    rewrite_by_lua_file /usr/example/lua/test_rewrite_2.lua;
    echo "rewrite2 uri : $uri, a : $arg_a";
}
```

第 2 步：编写 test\_rewrite\_2.lua 文件。

```
if ngx.req.get_uri_args()["jump"] == "1" then
    ngx.req.set_uri("/lua_rewrite_3", false);
    ngx.req.set_uri("/lua_rewrite_4", false);
    ngx.req.set_uri_args({a = 1, b = 2});
end
```

- `ngx.req.set_uri(uri, false)`：用于内部重写 URI（可以带参数），等价于“`rewrite ^/lua_rewrite_3`”。通过配合 `if/else` 可以实现“`rewrite ^/lua_rewrite_3 break;`”这种功能。此处两者都是 location 内部 URI 重写，不会重新发起新的 location 匹配。
- `ngx.req.set_uri_args`：重写请求参数，可以是字符串 (`a=1&b=2`)，也可以是表。

访问如 `http://192.168.1.2/lua_rewrite_2?jump=0` 时得到响应：

```
rewrite2 uri : /lua_rewrite_2, a :
```

访问如 `http://192.168.1.2/lua_rewrite_2?jump=1` 时得到响应：

```
rewrite2 uri : /lua_rewrite_4, a : 1
```

示例 3：使用 set\_uri 强制发起新的 location 实现重写跳转。

第 1 步，修改 example.conf 配置文件。

```
location /lua_rewrite_3 {
    default_type "text/html";
    rewrite_by_lua_file /usr/example/lua/test_rewrite_3.lua;
    echo "rewrite3 uri : $uri";
}
```

第 2 步，编写 test\_rewrite\_3.lua 文件。

```
if ngx.req.get_uri_args()["jump"] == "1" then
    ngx.req.set_uri("/lua_rewrite_4", true);
    ngx.log(ngx.ERR, "=====");
    ngx.req.set_uri_args({a = 1, b = 2});
end
```

ngx.req.set\_uri(uri, true)：可以内部重写 URI，即会发起新的匹配 location 请求，等价于“rewrite ^ /lua\_rewrite\_4 last;”，此处看 error log 是看不到我们记录的 log 的，因为 set\_uri(uri, true) 后发起新的 location 已经跳出当前代码了。

所以请求如 http://192.168.1.2/lua\_rewrite\_3?jump=1 会到新的 location 中得到响应，新的 location 是 /lua\_rewrite\_4。

rewrite 指令和 ngx.req.set\_uri 函数对应关系如下：

```
rewrite ^ /lua_rewrite_3;           等价于 ngx.req.set_uri("/lua_rewrite_3", false);。
rewrite ^ /lua_rewrite_3 break;     等价于 ngx.req.set_uri("/lua_rewrite_3", false);
加 if/else 判断 /break/return。
rewrite ^ /lua_rewrite_4 last;      等价于 ngx.req.set_uri("/lua_rewrite_4", true);。
```

---

**注意：**在使用 rewrite\_by\_lua 时，开启 rewrite\_log on; 后也看不到相应的 rewrite log。

---

## 10.4.5 access\_by\_lua

access\_by\_lua 用于访问控制。例如，如果只允许内网 IP 访问，可以使用如下形式：

```
allow 127.0.0.1;
allow 10.0.0.0/8;
allow 192.168.0.0/16;
allow 172.16.0.0/12;
deny all;
```

示例 4：实现使用 token 作为权限检查的应用。

第 1 步：修改 example.conf 配置文件。

```
location /lua_access {
    default_type "text/html";
    access_by_lua_file /usr/example/lua/test_access.lua;
    echo "access";
}
```

第 2 步：编写 test\_access.lua 文件。

```
if ngx.req.get_uri_args()["token"] ~= "123" then
    return ngx.exit(403)
end
```

如果访问如 `http://192.168.1.2/lua_access?token=234` 将得到 403 Forbidden 的响应，这样我们可以根据如“cookie/ 用户 token”来决定是否有访问权限。

### 10.4.6 content\_by\_lua

content\_by\_lua 是应用最多的指令，大部分任务是在这个阶段完成的，其他的过程往往为这个阶段准备数据，正式处理基本都在本阶段。

示例 5：下面是一个 RESTful API 的例子，这是一个 location，配置在 http 配置块中。

第 1 步：配置 nginx.conf。

```
user root;
worker_processes 4;
worker_rlimit_nofile 100000;
```

```
error_log logs/error.log;
```

```
pid logs/nginx.pid;
```

```
events{
    use epoll;
    worker_connections 10000;
}
```

```
http{
    include mime.types;
    default_type text/html;
```

```
access_log off;
```

```
server_tokens off;
```

```
lua_shared_dict gvar 50m;
```

```
sendfile on;
```

```
tcp_nopush on;
```

```
tcp_nodelay on;
```

```
open_file_cache max=10240 inactive=60s;
```

```
open_file_cache_valid 80s;
```

```
open_file_cache_min_uses 1;
```

```
keepalive_timeout 0;
```

```
chunked_transfer_encoding off;
```

```
lua_package_path "/usr/local/lib/lua/5.1/?.lua;;";
```

```
init_worker_by_lua_file init.lua;
```

```

    }
}

```

第 2 步：实现 API 的 location。

```

location /mdp/getConfig {
    lua_need_request_body on;
    client_max_body_size 50k;
    client_body_buffer_size 50k;
    #access_by_lua_file access.lua;
    content_by_lua '
local gwId=ngx.var.arg_gwId
local vdata={}
local cJSON = require "cjson"

vdata["name"]="MMP"
vdata["version"]="v1.0"
vdata["session"]=1
vdata["command"]=8
vdata["flow"]=0
vdata["sequence"]=1

vdata["gwId"]=gwId
vdata["cfgData"]=cfgData

local jsonRequest=cJSON.encode(vdata)

local devCmdKey="get_config_request_" .. gwId
local resp = ngx.location.capture("/redis_set1?key=" .. devCmdKey, {method =
    ngx.HTTP_POST, body = jsonRequest})

local responseKey="get_config_response_" .. gwId
for i = 1, 10, 1 do
    resp = ngx.location.capture("/redis_get?key=" .. responseKey)

    if resp.status == ngx.HTTP_OK and resp.body then
        local parser = require "redis.parser"
        local res, typ = parser.parse_reply(resp.body)

        resp = ngx.location.capture("/redis_del?key=" .. responseKey)
        if res ~= nil then
            ngx.print(res)
            ngx.exit(200)
        end
    else
        break
    end
    ngx.sleep(1)
end
';
}

```

这个示例演示了接收到请求后从向 Redis 中写入了一个命令，然后等待回应。

另外，在使用 PCRE 进行正则匹配时需要注意正则的写法，具体规则请参考 <http://wiki.nginx.org/HttpLuaModule> 中的 Special PCRE Sequences 部分。其他注意事项可参阅官

方文档。

### 10.4.7 header\_filter\_by\_lua

本阶段用于设置应答消息的头部信息，下面是一个修改应答头的例子。

```
location / {
    proxy_pass http://mybackend;
    header_filter_by_lua 'ngx.header.Foo = "blah"';
}
```

### 10.4.8 body\_filter\_by\_lua

本阶段使用 Lua 代码定义一个应答包体过滤器，例如：

```
location / {
    proxy_pass http://mybackend;
    body_filter_by_lua 'ngx.arg[1] = string.upper(ngx.arg[1])';
}
```

当向 ngx.arg[1] 传递一个 nil 值或空字符串，不会有任何数据块向下游传递。同样地，可以在 ngx.arg[2] 中传递一个“eof”实现目的，例如：

```
location /t {
    echo hello world;
    echo hiya globe;

    body_filter_by_lua '
        local chunk = ngx.arg[1]
        if string.match(chunk, "hello") then
            ngx.arg[2] = true -- new eof
            return
        end

        -- just throw away any remaining chunk data
        ngx.arg[1] = nil
    '
}
```

GET /t 将得到如下输出：

```
hello world
```

### 10.4.9 log\_by\_lua

log\_by\_lua 用于在 log 请求处理阶段，用 Lua 代码处理日志，但并不替换原有 log 处理。

示例 6：下面是一个从 \$upstream\_response\_time 中汇集并产生平均数据的例子。

```
lua_shared_dict log_dict 5M;
```



```

server {
    location / {
        proxy_pass http://mybackend;

        log_by_lua '
            local log_dict = ngx.shared.log_dict
            local upstream_time = tonumber(ngx.var.upstream_response_time)

            local sum = log_dict:get("upstream_time-sum") or 0
            sum = sum + upstream_time
            log_dict:set("upstream_time-sum", sum)

            local newval, err = log_dict:incr("upstream_time-nb", 1)
            if not newval and err == "not found" then
                log_dict:add("upstream_time-nb", 0)
                log_dict:incr("upstream_time-nb", 1)
            end
        ',

    location = /status {
        content_by_lua_block {
            local log_dict = ngx.shared.log_dict
            local sum = log_dict:get("upstream_time-sum")
            local nb = log_dict:get("upstream_time-nb")

            if nb and sum then
                ngx.say("average upstream response time: ", sum / nb,
                    " (", nb, " reqs)")
            else
                ngx.say("no data yet")
            end
        }
    }
}

```

#### 10.4.10 balancer\_by\_lua\_block

balancer\_by\_lua\* 的代码可作为上游服务器的负载均衡器。

示例 7：在两个端口上创建服务，根据请求的参数进行简单的除 2 作为 hash 值，取出对应端口作为服务。

```

upstream backend{
    server 0.0.0.0;
    balancer_by_lua_block{
        local balancer = require 'ngx.balancer'
        local port = {8088, 8089}
        local backend = ""
        local dvid = ngx.req.get_uri_args()["david"] or 0
        ngx.log(ngx.ERR, "dvid=", dvid)
        local hash = (dvid % 2) + 1
        ngx.log(ngx.ERR, "hash=" , hash)
        backend = port[hash]
    }
}

```

```

ngx.log(ngx.ERR, "backend=", backend)
ngx.log(ngx.ERR, "dvid=", dvid, " hash=", hash, " up=", backend)
local ok, err = balancer.set_current_peer("127.0.0.1", backend)

if not ok then
    ngx.log(ngx.ERR, "failed to set the current peer:", err)
    return ngx.exit(500)
end

ngx.log(ngx.DEBUG, " current peer ", backend)
}
}

```

## 10.5 小结

本章介绍了 Nginx 下 Lua 的实现原理，介绍了 Nginx 核心对 HTTP 请求划分的 11 个处理阶段；介绍了 ngx\_lua 模块参与处理的阶段，并对重要阶段进行了介绍，给出了简单示例，演示各阶段的主要功能。

Lua 的阶段处理，都是通过在 nginx.conf 配置文件中配置，注册 Lua 代码实现业务逻辑的。通过本章的学习，读者可以在自己的 nginx.conf 中开始尝试编写 Lua 代码了。



#### 第四部分 *Part 4*

## Nginx Lua 开发实战

- 第 11 章 Redis 操作
- 第 12 章 MySQL 操作
- 第 13 章 Memcached 操作
- 第 14 章 PostgreSQL 操作
- 第 15 章 MongoDB 操作
- 第 16 章 bit 库的使用
- 第 17 章 lfs 库的使用
- 第 18 章 resty.http 库的使用
- 第 19 章 lcurl 库的使用
- 第 20 章 FFI 扩展 C 库
- 第 21 章 cJSON 库的使用
- 第 22 章 lua-resty-template 类的使用
- 第 23 章 WebSocket 的使用
- 第 24 章 TCP 私有服务器实例
- 第 25 章 WebSocket 接入服务器实战
- 第 26 章 Nginx 应用简述

## Redis 操作

Redis 在系统中经常作为数据缓存、内存数据库使用，在大型系统中扮演着非常重要的作用。在 Nginx 核心系统中，Redis 是常备组件。本章主要介绍常用的 Redis 操作方法。

### 11.1 Redis 操作方法概述

Nginx 支持 3 种方法访问 Redis。

- HttpRedis 模块
- HttpRedis2Module
- lua-resty-redis 库

这 3 种方法的特点如下：

- HttpRedis 模块提供的指令少，功能单一，适合做简单缓存，可能以后会扩展。
- HttpRedis2Module 模块比 HttpRedis 模块操作更灵活，功能更强大。
- Lua-resty-redis 库是 OpenResty 提供的一个操作 Redis 的接口库，可根据自己的业务情况做一些逻辑处理，类似 PHP 开发中的各种扩展，适合做复杂的业务逻辑。

这 3 个模块都在 OpenResty 中集成了，前两个模块默认是启用的，对于第三个模块，可以使用 `--with-luajit` 开启 luajit 支持，lua-resty-redis 库默认是启用的。

关于这 3 种方法的选用，一般有如下建议：

- HttpRedis2Module 支持 Redis 2.0 协议，支持操作多。模块会原封不动地返回 Redis 服务的返回值。
- HttpRedis 只支持 Redis 的 `get` 命令和 `select` 命令，所以在使用 `set` 等操作时，本模块

无法支持，只适用于 get 和 select 操作的场景。一般需要对 Redis 的返回值做一定的解析。

- lua-resty-redis 更适合 Lua 包装数据，也更节省内存空间。

下面将具体介绍这 3 种方法，读者在自己的项目中可根据需要选择适合自己的方法。

## 11.2 HttpRedis 访问方法

HttpRedis 是 Nginx 的一个第三方扩展模块，用于实现 Nginx 直接使用 Redis 作为缓存系统。

### 11.2.1 示例

下面是一个通过 HttpRedis 使用 Redis 的例子。

#### 1) 配置 nginx.conf:

```
worker_processes 1;
error_log logs/error.log debug;

events {
    worker_connections 1024;
}

http {
    include mime.types;
    default_type application/octet-stream;
    sendfile on;
    keepalive_timeout 65;

    server {
        listen 80;
        server_name localhost;
        root html;
        index index.html index.htm;

        location / {
            default_type text/html;
            set $redis_key $uri;
            redis_pass 127.0.0.1:6379;
            error_page 404 = @fetch;
        }

        location @fetch {
            root html;
        }
    }
}
```

2) 测试配置并重启 Nginx。使用 `-s reload` 命令使 Nginx 重载配置文件:

```
/usr/local/openresty/nginx/sbin/nginx -p /usr/local/openresty/nginx/ -s reload
```

3) 在 `html` 目录下创建一个 `common.js` 文件, 内容为 `// this is a test js file`。命令如下:

```
cd /usr/local/openresty/nginx/html
echo '// this is a test js file'>common.js
```

4) 测试。使用 `curl` 命令访问测试路径, 查看测试结果:

```
#curl -i localhost/common.js
```

返回结果如下:

```
HTTP/1.1 200 OK
Server: ngx_openresty/1.2.3.14
Date: Wed, 20 Feb 2016 19:14:57 GMT
Content-Type: application/x-javascript
Content-Length: 23
Last-Modified: Wed, 20 Feb 2016 9:28:15 GMT
Connection: keep-alive
Accept-Ranges: bytes
```

```
// this is a test js file
```

因为 Redis 缓存是空的, 所以输出 404 状态码, 转向命名 location `@fetch`, 从本地文件系统读取 `/common.js` 文件, 本地是存在这个文件的, 所以返回了文件内容 `"// this is a test js file"`, 同时返回 HTTP 状态码 200。

下面我们在 Redis 里存入这个 key:

```
# redis-cli
redis 127.0.0.1:6379> set (/common.js) (/ fetch from redis)
OK
redis 127.0.0.1:6379> keys *
1) "/common.js"
```

再次请求:

```
# curl -i localhost/common.js
HTTP/1.1 200 OK
Server: ngx_openresty/1.2.3.14
Date: Wed, 20 Feb 2016 19:27:11 GMT
Content-Type: application/x-javascript
Content-Length: 19
Connection: keep-alive

// fetch from redis
```

结果跟预定流程一样。

## 11.2.2 HttpRedis API

HttpRedis 模块只使用 Redis 的 `get` 命令和 `select` 命令。

样例配置文件如下：

```
server {
    location / {
        set $redis_key $uri;

        redis_pass     name:6379;
        default_type    text/html;
        error_page       404 = /fallback;
    }

    location = /fallback {
        proxy_pass backend;
    }
}
```

下面为常用配置指令的详细介绍。

### 1. redis\_pass

语法：

```
redis_pass [name:port]
```

默认：none。

上下文：http、server、location。

说明：端点需要在 Redis 中首先设置数据，key 为 /uri?args。

### 2. redis\_bind

语法：

```
redis_bind [addr]
```

默认：none。

上下文：http、server、location。

说明：使用下面的地址作为 Redis 连接的源地址。

### 3. redis\_connect\_timeout

语法：

```
redis_connect_timeout [time]
```

默认：60000。

上下文：http、server、location。

说明：以毫秒为单位的 Redis 连接超时值。

### 4. redis\_read\_timeout

语法：

```
redis_read_timeout [time]
```

默认：60000。

上下文: http、server、location。

说明: 以毫秒为单位的 Redis 读取超时值。

#### 5. redis\_send\_timeout

语法:

```
redis_send_timeout [time]
```

默认: 60000。

上下文: http、server、location。

说明: 发送的超时值。

#### 6. redis\_buffer\_size

语法:

```
redis_buffer_size [size]
```

默认: see getpagesize(2)。

上下文: http、server、location。

说明: 以字节为单位的发送和接收缓冲区尺寸。

#### 7. redis\_next\_upstream

语法:

```
redis_next_upstream [error] [timeout] [invalid_response] [not_found] [off]
```

默认: error timeout。

上下文: http、server、location。

说明: 定义哪个失败条件可以导引请求向下一个 upstream 内 server 项流转。只有在 redis\_pass 中使用的 upstream 中有两个以上 server 的情况下才有效。

#### 8. redis\_gzip\_flag

语法:

```
redis_gzip_flag [number]
```

默认: unset。

上下文: location。

说明: 使用 gzip 压缩标志。

### 11.2.3 HttpRedis 变量

配置中如果有需要使用变量传递参数, 目前支持下面两个变量。

1) \$redis\_key: key 值。

2) \$redis\_db: Redis 数据库数量 (详情查看官方文档, 实用性不高)。



## 11.3 HttpRedis2Module 访问方法

HttpRedis2Module 是一个支持 Redis 2.0 模块协议的 Nginx 上游模块，提供非阻塞方式访问机制。

### 11.3.1 示例

下面例子演示 HttpRedis2Module 的使用方法。

1) 配置 nginx.conf:

```
worker_processes 1;
error_log logs/error.log debug;

events {
    worker_connections 1024;
}

http {
    include mime.types;
    default_type application/octet-stream;
    sendfile on;
    keepalive_timeout 65;

    server {
        listen 80;
        server_name localhost;
        root html;
        index index.html index.htm;

        location /get {
            set_unescape_uri $key $arg_key;
            redis2_query get $key;
            redis2_pass 127.0.0.1:6379;
        }

        location /set {
            set_unescape_uri $key $arg_key;
            set_unescape_uri $val $arg_val;
            redis2_query set $key $val;
            redis2_pass 127.0.0.1:6379;
        }
    }
}
```

2) 测试配置并重启 Nginx。命令如下:

```
/usr/local/openresty/nginx/sbin/nginx -p /usr/local/openresty/nginx/ -s reload
```

3) 测试。命令如下:

```
# curl -i localhost/get?key=/common.js
HTTP/1.1 200 OK
Server: ngx_openresty/1.2.3.14
```

```

Date: Wed, 20 Feb 2016 9:23:57 GMT
Content-Type: application/octet-stream
Transfer-Encoding: chunked
Connection: keep-alive

$19
// fetch from redis

# curl -i 'localhost/set?key=/common.js&val=set by nginx'
HTTP/1.1 200 OK
Server: ngx_openresty/1.2.3.14
Date: Wed, 20 Feb 2016 9:23:41 GMT
Content-Type: application/octet-stream
Transfer-Encoding: chunked
Connection: keep-alive

+OK

# curl -i localhost/get?key=/common.js
HTTP/1.1 200 OK
Server: ngx_openresty/1.2.3.14
Date: Wed, 20 Feb 2016 9:23:45 GMT
Content-Type: application/octet-stream
Transfer-Encoding: chunked
Connection: keep-alive

$12
set by nginx

```

结果与预期一致，其中的 \$19 和 \$12 是返回的数据长度。

### 11.3.2 nginx.conf 配置

HTTPRedis2Module 通过配置指令实现 Redis 访问功能，它是一个上游模块，下面是样例配置文件。

```

location = /foo {
    set $value 'first';
    redis2_query set one $value;
    redis2_pass 127.0.0.1:6379;
}

# GET /get?key=some_key
location = /get {
    set_unescape_uri $key $arg_key; # this requires ngx_set_misc
    redis2_query get $key;
    redis2_pass foo.com:6379;
}

# GET /set?key=one&val=first%20value
location = /set {
    set_unescape_uri $key $arg_key; # this requires ngx_set_misc
    set_unescape_uri $val $arg_val; # this requires ngx_set_misc
    redis2_query set $key $val;
}

```

```

    redis2_pass foo.com:6379;
}

# multiple pipelined queries
location = /foo {
    set $value 'first';
    redis2_query set one $value;
    redis2_query get one;
    redis2_query set one two;
    redis2_query get one;
    redis2_pass 127.0.0.1:6379;
}

location = /bar {
    # $ is not special here...
    redis2_literal_raw_query '*1\r\n$4\r\nping\r\n';
    redis2_pass 127.0.0.1:6379;
}

location = /bar {
    # variables can be used below and $ is special
    redis2_raw_query 'get one\r\n';
    redis2_pass 127.0.0.1:6379;
}

# GET /baz?get%20foo%0d%0a
location = /baz {
    set_unescape_uri $query $query_string; # this requires the ngx_set_misc module
    redis2_raw_query $query;
    redis2_pass 127.0.0.1:6379;
}

location = /init {
    redis2_query del key1;
    redis2_query lpush key1 C;
    redis2_query lpush key1 B;
    redis2_query lpush key1 A;
    redis2_pass 127.0.0.1:6379;
}

location = /get {
    redis2_query lrange key1 0 -1;
    redis2_pass 127.0.0.1:6379;
}

```

### 11.3.3 常用指令

#### 1. redis2\_query

语法:

```
redis2_query cmd arg1 arg2 ...
```

默认: no。

上下文: location、location if。

说明: 定义一个 Redis 命令行, 与 redis-cli 操作非常相似。可以定义多个指令, 这些指令会被流水线执行, 例如:

```
location = /pipelined {
    redis2_query set hello world;
    redis2_query get hello;

    redis2_pass 127.0.0.1:$TEST_NGINX_REDIS_PORT;
}
```

通过 GET 访问 /pipelined 会得到两个原始的 Redis 应答。

```
+OK
$5
world
```

新行是 CR LF (\r\n)。

## 2. redis2\_raw\_query

语法:

```
redis2_raw_query QUERY
```

默认: no。

上下文: location、location if。

说明: 定义一个原始 Redis 查询, 可以使用 Nginx 变量作为查询参数。

只允许一个查询命令, 否则会收到错误信息。如果要使用多个流水线式查询命令, 可使用下条指令。

## 3. redis2\_raw\_queries

语法:

```
redis2_raw_queries N QUERIES
```

默认: no。

上下文: location、location if。

说明: 在查询参数中定义多条命令。可以使用 Nginx 变量。

例如:

```
location = /pipelined {
    redis2_raw_queries 3 "flushall\r\nget key1\r\nget key2\r\n";
    redis2_pass 127.0.0.1:6379;
}
```

```
# GET /pipelined2?n=2&cmds=flushall%0D%0Aget%20key%0D%0A
location = /pipelined2 {
    set_unescape_uri $n $arg_n;
    set_unescape_uri $cmds $arg_cmds;
```

```

redis2_raw_queries $n $cmds;

redis2_pass 127.0.0.1:6379;
}

```

注意，第二个例子中的 `set_unescape_uri` 指令是由 `set-misc-nginx-module` 模块提供的。

#### 4. redis2\_literal\_raw\_query

语法：

```
redis2_literal_raw_query QUERY
```

默认：no。

上下文：location、location if。

说明：指定一个原始查询，但不能使用 Nginx 变量。也就是说，可以在查询参数中使用 \$ 符号。只能使用一个查询命令。

#### 5. redis2\_pass

语法：

```

redis2_pass <upstream_name>
redis2_pass <host>:<port>

```

默认：no。

上下文：location、location if。

阶段：content。

说明：指定 Redis 服务器端点。

#### 6. redis2\_connect\_timeout

语法：

```
redis2_connect_timeout <time>
```

默认：60s。

上下文：http、server、location。

说明：连接到服务器的超时值，默认为 60 秒。

#### 7. redis2\_send\_timeout

语法：

```
redis2_send_timeout <time>
```

默认：60s。

上下文：http、server、location。

说明：发送超时值。

#### 8. redis2\_read\_timeout

语法：

```
redis2_read_timeout <time>
```

默认: 60s。

上下文: http、server、location。

说明: 读取超时值。

### 9. redis2\_buffer\_size

语法:

```
redis2_buffer_size <size>
```

默认: 4k/8k。

上下文: http、server、location。

说明: 读取应用的缓冲区大小, 不需要和可能收到的最大应答包大小一致。默认尺寸是页尺寸, 4k 或 8k。

### 10. redis2\_next\_upstream

语法:

```
redis2_next_upstream [ error | timeout | invalid_response | off ]
```

默认: error timeout。

上下文: http、server、location。

说明: 指定哪个条件可以使请求向下个 upstream server 流转。当然只在 redis2\_pass 中的 upstream 中有两个以上 server 的情况下有效。

例如:

```
upstream redis_cluster {
    server 127.0.0.1:6379;
    server 127.0.0.1:6380;
}

server {
    location = /redis {
        redis2_next_upstream error timeout invalid_response;
        redis2_query get foo;
        redis2_pass redis_cluster;
    }
}
```

## 11.3.4 技术点

HttpRedis2Module 使用中有若干个需要注意的技术点。

### 1. 连接池

可以使用 HttpUpstreamKeepaliveModule 为 Redis 提供 TCP 连接池 (Connection Pool)。

例如:

```

http {
    upstream backend {
        server 127.0.0.1:6379;

        # a pool with at most 1024 connections
        # and do not distinguish the servers:
        keepalive 1024;
    }

    server {
        ...
        location = /redis {
            set_unescape_uri $query $arg_query;
            redis2_query $query;
            redis2_pass backend;
        }
    }
}

```

## 2. 选择 Redis 数据库

Redis 提供 select 命令切换 Redis 数据库，可以在 redis2\_query 中使用，例如：

```
redis2_query select 8;redis2_query get foo;
```

## 3. Lua 协同能力

本模块可以作为 lua-ngx-module 的非阻塞 redis2 客户端，下面是一个使用 get 子请求的示例：

```

location = /redis {
    internal;

    # set_unescape_uri is provided by ngx_set_misc
    set_unescape_uri $query $arg_query;

    redis2_raw_query $query;
    redis2_pass 127.0.0.1:6379;
}

location = /main {
    content_by_lua '
        local res = ngx.location.capture("/redis",{ args = { query = "ping\\r\\n" } })
        ngx.print("[ " .. res.body .. " ]")
    '
}

```

访问 /main:

```
[+PONG\r\n]
```

当把内嵌的 Lua 代码保存到 .lua 文件时，需要对 \r\n 转义。也可以使用 post/put 子请求通过请求包体传输原始请求，这样就不需要进行请求 URI 转码了，节省 CPU 资源。

```
location = /redis {
```

```

        internal;
        # $echo_request_body is provided by the ngx_echo module
        redis2_raw_query $echo_request_body;
        redis2_pass 127.0.0.1:6379;
    }

    location = /main {
        content_by_lua '
            local res = ngx.location.capture("/redis",{ method = ngx.HTTP_PUT, body
                = "ping\\r\\n" })
            ngx.print "[" .. res.body .. "]"
        '
    }

```

这个例子的输出和 GET 的示例输出相同。

可以使用 Lua 在复杂的 hash 规则下选择固定的 Redis 端点:

```

upstream redis-a {
    server foo.bar.com:6379;
}

upstream redis-b {
    server bar.baz.com:6379;
}

upstream redis-c {
    server blah.blah.org:6379;
}

server {
    ...

    location = /redis {
        set_unescape_uri $query $arg_query;
        redis2_query $query;
        redis2_pass $arg_backend;
    }

    location = /foo {
        content_by_lua "
            -- pick up a server randomly
            local servers = {'redis-a', 'redis-b', 'redis-c'}
            local i = ngx.time() % #servers + 1
            local srv = servers[i]
            local res = ngx.location.capture('/redis',{args={query='...',
                backend = srv }})
            ngx.say(res.body)
        "
    }
}

```

#### 4. 流水线请求

下面是一个多请求的例子:

```

location = /redis2 {
    internal;

```



```

    redis2_raw_queries $args $echo_request_body;
    redis2_pass 127.0.0.1:6379;
}

```

```

location = /test {
    content_by_lua_file conf/test.lua;
}

```

URI 的 args 传输请求数量，包体传递请求字符串。

创建 conf/test.lua 文件：

```

-- conf/test.lua
local parser = require "redis.parser"

local reqs = {
    {"set", "foo", "hello world"},
    {"get", "foo"}
}

local raw_reqs = {}
for i, req in ipairs(reqs) do
    table.insert(raw_reqs, parser.build_query(req))
end

local res = ngx.location.capture("/redis2?" .. #reqs,
    { body = table.concat(raw_reqs, "") })

if res.status ~= 200 or not res.body then
    ngx.log(ngx.ERR, "failed to query redis")
    ngx.exit(500)
end

local replies = parser.parse_replies(res.body, #reqs)
for i, reply in ipairs(replies) do
    ngx.say(reply[1])
end

```

使用 curl 访问 /test 将得到下面输出：

```

OK
hello world

```

## 5. redis publish/subscribe 支持

本模块有限支持 publish/subscribe 性能，这是因为 REST 和 HTTP 是无状态的。考虑下面的例子：

```

location = /redis {
    redis2_raw_queries 2 "subscribe /foo/bar\r\n";
    redis2_pass 127.0.0.1:6379;
}

```

然后通过 redis-cli 命令行对 key /foo/bar 发布一个消息，将从 /redis 接收到两个回复，这两个回复也可以解析。

## 6. publish/subscribe 限制

使用 publish/subscribe 有下面的限制:

- 不能在 HttpUpstreamKeepaliveModule 中以 Redis 作为 upstream, 因为只有短的 Redis 连接可以工作。
- Redis 为 Nginx 的 error.log 产生额外的警告数据机制, 会有一些竞争的情况产生, 虽然这种警告的产生是比较少见的。
- 需要调整多个超时值, 如 redis2\_connect\_timeout 和 redis2\_read\_timeout。

如果不能接受这些限制, 推荐使用 lua-resty-redis 库。

## 7. 性能调整

在模块使用中, 可以使用下列性能优化机制。

- 确保使用 TCP 连接池 (HttpUpstreamKeepaliveModule 提供), 可能的情况下使用流水线指令。
- 在多核的机器上使用多个 Redis 服务示例。
- 当使用诸如 ab 或 http\_load 时, 要确保 error 日志级别足够高 (如 warn), 避免工作进程频繁刷写日志文件。

### 11.3.5 应答包解析

应答包需要使用 redis.parser 进行解析, 如果 value 为原始数据, 则解析出来的数据直接可用, 如果 value 保存进去就为 JSON 格式, 则需要使用 CJSON 进行进一步解析处理:

```
...
local json_data
local cJSON = require "cjson"
local parser = require "redis.parser"
local res, err
local resp

resp = ngx.location.capture("/redis_get?key=" .. "123")
if resp.status == ngx.HTTP_OK and resp.body then
    local res, typ = parser.parse_reply(resp.body)
    if res ~= nil then
        -- 如果是原始格式, 则此处 res 可以直接使用, 如果原始数据是 JSON, 则往下进行 JSON 解析处理
        local ev={}
        ev=cjson.decode(res)
        local sid = ev["sid"]
        local ev_time = ev["ev_time"]
        local ev_type = ev["ev_type"]
    end
end
.....
```

## 11.4 lua-resty-redis 访问方法

lua-resty-redis 是 OpenResty 团队写的组件, 相比其他 Redis 库, 本库占用内存更小, 使用更灵活, 功能更强大, 适合在复杂应用下使用。如果只是简单访问 Redis 数据, 使用 HttpRedis2Module 可能更简洁一些, 所以, 需要根据应用场合选用库。

### 11.4.1 示例

下面是一个 lua-resty-redis 库完整的使用范例。代码编写在 nginx.conf 中。

# 如果使用 OpenResty, 则不需要下面的配置行。

```
lua_package_path "/path/to/lua-resty-redis/lib/?lua;;";
```

```
server {
    location /test {
        content_by_lua_block {
            local redis = require "resty.redis"
            local red = redis:new()

            red:set_timeout(1000) -- 1 sec

            -- or connect to a unix domain socket file listened
            -- by a redis server:
            -- local ok, err = red:connect("unix:/path/to/redis.sock")

            local ok, err = red:connect("127.0.0.1", 6379)
            if not ok then
                ngx.say("failed to connect: ", err)
                return
            end

            ok, err = red:set("dog", "an animal")
            if not ok then
                ngx.say("failed to set dog: ", err)
                return
            end

            ngx.say("set result: ", ok)

            local res, err = red:get("dog")
            if not res then
                ngx.say("failed to get dog: ", err)
                return
            end

            if res == ngx.null then
                ngx.say("dog not found.")
                return
            end

            ngx.say("dog: ", res)
```

```

red:init_pipeline()
red:set("cat", "Marry")
red:set("horse", "Bob")
red:get("cat")
red:get("horse")
local results, err = red:commit_pipeline()
if not results then
    ngx.say("failed to commit the pipelined requests: ", err)
    return
end

for i, res in ipairs(results) do
    if type(res) == "table" then
        if res[1] == false then
            ngx.say("failed to run command ", i, ": ", res[2])
        else
            -- process the table value
        end
    else
        -- process the scalar value
    end
end

-- put it into the connection pool of size 100,
-- with 10 seconds max idle time
local ok, err = red:set_keepalive(10000, 100)
if not ok then
    ngx.say("failed to set keepalive: ", err)
    return
end

-- or just close the connection right away:
-- local ok, err = red:close()
-- if not ok then
--     ngx.say("failed to close: ", err)
--     return
-- end
end
}
}
}

```

## 11.4.2 API 函数

### 1. API 概述

在 lua-resty-redis 中, 所有的 Redis 命令都有自己的方法, 方法名字和命令名字相同, 只是全部为小写。

命令参数可以直接填充在方法调用中。例如, “GET” 命令接受一个 key 参数, 可以这样调用 “get” 方法:

```
local res, err = red:get("key")
```

类似地, “LRANGE” 命令接受 3 个参数, 则可以这样调用 “lrange”:

```
local res, err = red:lrange("nokey", 0, 1)
```

更多的例子如下:

```
-- HMGET myhash field1 field2 nofield
local res, err = red:hget("myhash", "field1", "field2", "nofield")
-- HMSET myhash field1 "Hello" field2 "World"
local res, err = red:hset("myhash", "field1", "Hello", "field2", "World")
```

这些方法成功时返回 success, 失败返回 nil 和保存在 err 中的错误描述。

- A Redis “status reply” 结果返回一个前导为 “+” 的字符串。
- A Redis “integer reply” 返回一个 Lua 数据型数据。
- A Redis “error reply” 返回一个 false 值和一个错误描述字符串。
- A non-nil Redis “bulk reply” 返回一个 Lua 表。如果错误, 将返回一个包含两个元素的表 {false, err}。
- A nil multi-bulk reply 返回 Lua ngx.null 值。

下面将详细介绍 lua-resty-redis 提供的 API。

## 2. API 详述

lua-resty-redis 提供了访问 Redis 的详细 API, 包括创建对接、连接、操作、数据处理等。这些 API 基本上与 Redis 的操作一一对应。

### (1) new

语法:

```
red, err = redis:new()
```

说明: 创建一个 Redis 对象。

### (2) connect

语法:

```
ok, err = red:connect(host, port, options_table?)
ok, err = red:connect("unix:/path/to/unix.sock", options_table?)
```

说明: 连接到远程的 Redis 服务器。在进行主机解析之前, 本方法会先遍历连接池。可选的表用于向方法传送连接参数。

pool 指声明一个自定义连接池名字, 如果未指定, 将使用模板 <host>:<port> or <unix-socket-path>。

### (3) set\_timeout

语法:

```
red:set_timeout(time)
```

说明: 设置子请求操作的超时值, 单位为毫秒, 包括 connect 方法。

#### (4) set\_keepalive

语法:

```
ok, err = red:set_keepalive(max_idle_timeout, pool_size)
```

说明: 把当前 Redis 连接立即放入连接池。

可以指定最大空闲时间, 以及工作进程级的连接池大小。如果成功则返回 1, 失败则返回 nil 和错误描述。

本方法要放在原本放 close 方法的地方, 本方法被调用, Redis 对象马上就进入 close 状态, 后续操作将会失败。

#### (5) get\_reused\_times

语法:

```
times, err = red:get_reused_times()
```

说明: 获取本连接是否是重用的连接。如果失败则返回 nil 和字符串描述。如果当前连接是不得重用的连接池连接, 则总是返回 0。

#### (6) close

语法:

```
ok, err = red:close()
```

说明: 关闭当前连接, 成功则返回 1, 失败则返回 nil 和错误描述。

#### (7) init\_pipeline

语法:

```
red:init_pipeline()  
red:init_pipeline(n)
```

说明: 使能 Redis 流水线模式。所有 Redis 调用方法自动缓存起来, 当 commit\_pipeline 方法调用被提交, 或 cancel\_pipeline 调用时被取消。

本方法总是成功。如果对象已经在流水线模式中, 再次调用将会清空缓存, 丢失之前未发送的查询命令。

可靠参数 n 定义可以加到流水线中的命令数量。

#### (8) commit\_pipeline

语法:

```
results, err = red:commit_pipeline()
```

说明: 提交缓存的查询命令, 并退出流水线模式。这些请求的应答将被自动收集, 以 multi-bulk 类型应答给上层。

#### (9) cancel\_pipeline

语法:

```
red:cancel_pipeline()
```

说明：退出流水线模式，并丢弃所有的缓存命令。这个方法总是成功。如果 Redis 对象不是流水线模式，也可以成功，只是程序空跑。

#### (10) hmset

语法：

```
red:hmset(myhash, field1, value1, field2, value2, ...)
red:hmset(myhash, { field1 = value1, field2 = value2, ... })
```

说明：为“hmset”命令定义一个封装。当只有 3 个参数（包括 red 对象）时，最后一个参数必须是一个 Lua 表，存放 field/value 对。

#### (11) array\_to\_hash

语法：

```
hash = red:array_to_hash(array)
```

说明：辅助函数，转换数据风格 Lua 表到 hash 表。

#### (12) read\_reply

语法：

```
res, err = red:read_reply()
```

说明：从服务器读取一个应答。这个方法对于 pub/sub 非常有用。例如：

```
local cJSON = require "cjson"
local redis = require "resty.redis"
```

```
local red = redis:new()
local red2 = redis:new()
```

```
red:set_timeout(1000) -- 1 sec
red2:set_timeout(1000) -- 1 sec
```

```
local ok, err = red:connect("127.0.0.1", 6379)
if not ok then
    ngx.say("1: failed to connect: ", err)
    return
end
```

```
ok, err = red2:connect("127.0.0.1", 6379)
if not ok then
    ngx.say("2: failed to connect: ", err)
    return
end
```

```
local res, err = red:subscribe("dog")
if not res then
    ngx.say("1: failed to subscribe: ", err)
    return
end
```

```

ngx.say("1: subscribe: ", cJSON.encode(res))

res, err = red2:publish("dog", "Hello")
if not res then
    ngx.say("2: failed to publish: ", err)
    return
end

ngx.say("2: publish: ", cJSON.encode(res))

res, err = red:read_reply()
if not res then
    ngx.say("1: failed to read reply: ", err)
    return
end

ngx.say("1: receive: ", cJSON.encode(res))

red:close()
red2:close()

```

运行这个例子可以得到如下输出:

```

1: subscribe: ["subscribe","dog",1]
2: publish: 1
1: receive: ["message","dog","Hello"]

```

### 11.4.3 技术点

使用 lua-resty-redis 库时也有若干技术点需要注意, 下面将分别进行介绍。

#### 1. Redis 验证

Redis 使用 AUTH 命令验证, 例如:

```

local redis = require "resty.redis"
local red = redis:new()

red:set_timeout(1000) -- 1 sec

local ok, err = red:connect("127.0.0.1", 6379)
if not ok then
    ngx.say("failed to connect: ", err)
    return
end

local res, err = red:auth("foobared")
if not res then
    ngx.say("failed to authenticate: ", err)
    return
end

```

假设 Redis 服务器在 redis.conf 配置了密码 foobared:

```
requirepass foobared
```



如果密码错误，运行上面的程序会得到下面的错误信息：

```
failed to authenticate: ERR invalid password
```

## 2. Redis 事务

本库支持 Redis 事务，例如：

```
local cJSON = require "cjson"
local redis = require "resty.redis"
local red = redis:new()

red:set_timeout(1000) -- 1 sec

local ok, err = red:connect("127.0.0.1", 6379)
if not ok then
    ngx.say("failed to connect: ", err)
    return
end

local ok, err = red:multi()
if not ok then
    ngx.say("failed to run multi: ", err)
    return
end

ngx.say("multi ans: ", cJSON.encode(ok))

local ans, err = red:set("a", "abc")
if not ans then
    ngx.say("failed to run sort: ", err)
    return
end
ngx.say("set ans: ", cJSON.encode(ans))

local ans, err = red:lpop("a")
if not ans then
    ngx.say("failed to run sort: ", err)
    return
end
ngx.say("set ans: ", cJSON.encode(ans))

ans, err = red:exec()
ngx.say("exec ans: ", cJSON.encode(ans))

red:close()
```

输出如下：

```
Then the output will be
multi ans: "OK"
set ans: "QUEUED"
set ans: "QUEUED"
exec ans: ["OK",[false,"ERR Operation against a key holding the wrong kind of
value"]]
```

### 3. 负载均衡和故障切换

可以在自己的 Lua 里实现一般的负载均衡逻辑。保存一张所有有效 Redis 端点的 Lua 表（如主机名或端口号），每个请求中按照某种规则选取一个服务器（轮循或 key 哈希）。

同样地，可以自己实现更灵活的故障切换规则。

### 4. 调试

通常我们使用 lua-cjson 库将返回值转换为 JSON 格式，例如：

```
local cjson = require "cjson"
...
local res, err = red:mget("h1234", "h5678")
if res then
    print("res: ", cjson.encode(res))
end
```

### 5. 自动错误日志

默认地，ngx\_lua 方法在套接字出现错误时会自动记录日志，如果已经自己做了日志处理，需要禁用自动记录错误日志功能：

```
lua_socket_log_errors off;
```

## 11.4.4 问题列表

lua-resty-redis 库属于比较常用的模块，下面是一些常见问题。

- 保证连接池容量配置正确（set\_keepalive）。注意，连接池是以工作线程为单位的。如果系统整体处理 1000 个请求，而工作进程有 10 个，则每个连接池容量应该是  $1000/10=100$ （个）。如果直接配成 1000，则实际池容量为 10 000，浪费内存。
- 保证 Redis 侧 backlog 设置为足够大。在 Redis 2.8 及以上版本中，可以直接在 redis.conf 上打开 tcp-backlogparameter（同时需要打开 Linux 内核参数 SOMAXCONN，进行相应配置）。也可能要在 redis.conf 中打开 maxclients。
- 保证 set\_timeout 方法设置的超时值不会太短。如果需要，在超时时重做失败的操作，并且关闭自动记录错误日志功能。
- 如果工作进程负载非常重，Nginx 的事件循环可能被 CPU 阻塞非常多。使用基于 C 或基于 Lua 的图形分析工具，对 CPU 进行优化。
- 如果工作进程 CPU 负载非常轻，则事件循环可能被一些系统调用阻塞了（如 I/O 系统调用）。可以运行 epoll-loop-blocking-distr 工具确认问题。如果确实是这个问题，可以使用图形化工具分析实际阻塞的位置。
- 如果 redis-server 是 100% 的 CPU 使用率，则需要考虑使用多个 Redis 端点进行负载均衡，或者使用图形化工具分析 Redis 内部瓶颈。

### 11.4.5 限制

使用 lua-resty-redis 库时，需要注意下面这些限制。

- 本库不能用到这些上下文：init\_by\_lua、set\_by\_lua、log\_by\_lua、header\_filter\_by\_lua，在这些地方，ngx\_lua cosocket API 是无效的。
- resty.redis 对象不能被存储在模块级的变量中（因为会被其他例程分享），要保存在局部变量或 ngx.ctxtable 中，这些地方的每个例程有自己的数据副本，不会出现访问冲突。

### 11.4.6 安装

OpenResty 包内建了访问模块，可以直接使用：

```
local redis = require "resty.redis"
...
```

如果使用自己的 Nginx+ngx\_lua 环境，需要配置 lua\_package\_path 指令，以确保工作进程的账号有权限访问这些 .lua 文件，具体指令如下：

```
# nginx.conf
http {
    lua_package_path "/path/to/lua-resty-redis/lib/?.lua;;";
    ...
}
```

## 11.5 小结

本章介绍了在 ngx\_lua 中访问 Redis 的 3 种方法，分别是 HttpRedis、HttpRedis2Module 和 lua-resty-redis。3 种方法各有适用场合，根据需要进行选择使用。本章对每种方法都进行了概述，给出了示例，并详细解释了各库提供的 API 以及使用的注意事项。示例中未涉及的 API 可参照描述加入示例，形成读者自己的使用代码。

## MySQL 操作

MySQL 是一个使用广泛的关系型数据库，用来保存关系型数据。根据应用，MySQL 规模可大可小。大的系统可以做读写分离的大型集群。

在 ngx\_lua 中，MySQL 有两种访问模式：

1) 使用 ngx\_lua 模块和 lua-resty-mysql 模块：这两个模块是安装 OpenResty 时默认安装的。

2) 使用 drizzle\_nginx\_module (HttpDrizzleModule) 模块：需要单独安装，这个库现在不在 OpenResty 中。

本章分别介绍这两种 MySQL 操作方法。

### 12.1 lua-resty-mysql 访问方式

lua-resty-mysql 是 OpenResty 开发的模块，使用灵活、功能强大，适合复杂的业务场景，同时支持存储过程的访问。

#### 12.1.1 示例

下面是一个 lua-resty-mysql 使用 MySQL 数据库的示例。

##### 1. 创建测试数据库

在安装好的 MySQL 中执行下面语句，创建表和初始数据：

```
mysql> create table users(id int,username varchar(30),age tinyint);
Query OK, 0 rows affected (0.00 sec)
```

```
mysql> insert into users values(1,'zhangsan',24);
Query OK, 1 row affected (0.00 sec)
```

```
mysql> insert into users values(2,'lisi',26);
Query OK, 1 row affected (0.00 sec)
```

## 2. 修改 Nginx 配置文件 nginx.conf

```
worker_processes      1;
events {
    worker_connections  1024;
}
http {
    include              mime.types;
    default_type         application/octet-stream;
    sendfile             on;
    keepalive_timeout    65;
    server {
        listen          80;
        server_name     localhost;
        root            html;
        index            index.html index.htm;
        location / {
            content_by_lua '
                local mysql = require "resty.mysql"
                local db,err = mysql:new()
                if not db then
                    ngx.say("failed to instantiate mysql: ",err)
                    return
                end

                db:set_timeout(1000)

                local ok,err,errno,sqlstate = db:connect{
                    host = "127.0.0.1",
                    port = 3306,
                    database = "test",
                    user = "root",
                    password = "",
                    max_package_size = 1024
                }
                if not ok then
                    ngx.say("failed to connect: ", err, ": ", errno, " ", sqlstate)
                    return
                end

                res,err,errno,sqlstate = db:query("select id,username,age from
                    users where id=1")

                if not res then
                    ngx.say("bad result: ", err, ": ", errno, ": ", sqlstate, ".")
                    return
                end

                local cJSON = require "cjson"
                ngx.say(cJSON.encode(res))
            '
        }
    }
}
```

```

    };
}
error_page 500 502 503 504 /50x.html;
location = /50x.html {
    root html;
}
}
}

```

从 MySQL 数据库返回的是 JSON 数据，需要使用 CJSON 进行转换处理。

也可以将上面的 Lua 代码放到一个以 .lua 为扩展名的文件里，并在 nginx.conf 中使用 content\_by\_lua\_file 指令引用。

### 3. 重启 Nginx 服务

命令如下：

```

# /usr/local/openresty/nginx/sbin/nginx -t -p /usr/local/openresty/nginx/conf/
nginx: the configuration file /usr/local/openresty/nginx/conf/nginx.conf syntax is
ok
nginx: configuration file /usr/local/openresty/nginx/conf/nginx.conf test is
successful

# /usr/local/openresty/nginx/sbin/nginx -p /usr/local/openresty/nginx/conf/ -s
reload

```

### 4. 测试

命令如下：

```

# curl localhost
[{"username":"zhangsan","age":24,"id":1}]

```

输出结果是 JSON 数据。

## 12.1.2 安装

如果使用 OpenResty 安装，直接使用即可：

```
local mysql = require "resty.mysql"
...

```

如果使用自己的 Nginx+ngx\_lua，需要配置 lua\_package\_path 指令添加 lua-resty-mysql 源树到 LUA\_PATH 搜索路径，例如：

```

# nginx.conf
http {
    lua_package_path "/path/to/lua-resty-mysql/lib/?.lua;;";
    ...
}

```

要保证运行 Nginx 工作进程的账号有权限访问 .lua 文件。

### 12.1.3 方法与函数

本节详述 lua-resty-mysql 提供的方法。方法涉及传输层连接池、超时时间配置等。

#### 1. new

语法：

```
db, err = mysql:new()
```

说明：创建一个 MySQL 连接对象，遇到错误时返回 db=nil 和存放在 err 中的错误描述。

#### 2. connect

语法：

```
ok, err = db:connect(options)
```

说明：尝试连接到一个远程的 MySQL 服务器。

这里 options 是一个参数的 Lua 表，涉及的具体参数如下：

- host：服务器主机名或 IP 地址。
- port：服务器监听端口，默认为 3306。
- path：UNIX 套接字文件存放路径。
- database：使用的数据库名，相当于“select database;”。
- user：登录的用户名。
- password：登录密码（明文）。
- max\_packet\_size：MySQL 服务器应答包最大长度（默认 1MB）。
- ssl：如果设置为 true，使用 SSL 连接到 MySQL（默认是 false），如果不支持 SSL 或禁用了 SSL，则返回“ssl disabled on server”错误信息。
- ssl\_verify：如果设置为 true，使用服务器的 SSL 证书校验（默认是 false）。注意，需要配置 lua\_ssl\_trusted\_certificate 指定 MySQL 服务器使用的 CA 证书。可能也需要配置 lua\_ssl\_verify\_depth。
- pool：MySQL 连接池的名称。如果没有指定，连接池名称将会被指定为 user:database:host:port 格式或 user:database:path。
- compact\_arrays：设置为 true 时，查询和读取方法将返回 array-of-arrays 结构的 resultset，默认是 array-of-hashes 结构。

本方法在进行主机名解析并连接到远程服务器前，总是在连接池中寻找之前调用创建的空闲连接。

#### 3. set\_timeout

语法：

```
db:set_timeout(time)
```

说明：设置子请求的超时时间（ms），包括 connect 方法。

#### 4. set\_keepalive

语法：

```
ok, err = db:set_keepalive(max_idle_timeout, pool_size)
```

说明：把当前的 MySQL 连接立即放进 ngx\_lua cosocket 连接池。可以指定最大空闲时间（ms），可以指定每一个 Nginx 工作进程池的最大容量。如果成功，则返回 1；如果错误，则返回 nil 和错误描述。

这个方法可以放到原本放 close 方法的地方。这个方法将把 resty.mysql 对象直接设置为关闭状态。所有的子操作都将返回关闭错误。

#### 5. get\_reused\_times

语法：

```
times, err = db:get_reused_times()
```

说明：返回当前连接的重用次数。如果有错误，则返回 nil 和错误描述。

如果当前连接不是来自于内建连接池，则总是返回 0，表示从未重用过。如果当前连接来自于连接池，则返回值总是非 0。所以，可以使用本方法判断连接是否来自于连接池。

#### 6. close

语法：

```
ok, err = db:close()
```

说明：关闭当前 MySQL 连接并返回状态。如果成功，则返回 1；如果出现任何错误，则将返回 nil 和错误描述。

#### 7. send\_query

语法：

```
bytes, err = db:send_query(query)
```

说明：异步向远程 MySQL 发送一个查询。如果成功则返回成功发送的字节数；如果错误，则返回 nil 和错误描述。

需要使用 read\_result 方法读取应答。

#### 8. read\_result

语法：

```
res, err, errcode, sqlstate = db:read_result()  
res, err, errcode, sqlstate = db:read_result(nrows)
```

说明：从 MySQL 服务器返回结果中读取一行数据。res 返回一个描述 OK 包或结果集包的 Lua 表。



返回值是一个容纳多行的数组。每一行是一个数据列的 key-value 对，例如：

```
{
  { name = "Bob", age = 32, phone = ngx.null },
  { name = "Marry", age = 18, phone = "10666372"}
}
```

如果查询没有返回值，则返回类似的内容：

```
{
  insert_id = 0,
  server_status = 2,
  warning_count = 1,
  affected_rows = 32,
  message = nil
}
```

如果当前结果返回的是多结果集，则 err 内容是 “again”。所以需要检测 err 是否是 again，直到把所有的数据读取完毕。该指令主要用于查询语句中有多个查询或多个查询语句的情况下。

任何错误，将返回最多 4 个值：nil、err、errcode 和 sqlstate。err 返回一个错误描述字符串，errcode 返回 MySQL 错误码，sqlstate 返回由 5 个字符组成的标准 SQL 错误码。注意，errcode 和 sqlstate 可能因为 MySQL 没返回它们而造成值为 nil。

可选的参数 nrows 可以用来指定返回结果集的最大值。这个参数可以用来预申请空调，默认这个值为 4。

## 9. query

语法：

```
res, err, errcode, sqlstate = db:query(query)
res, err, errcode, sqlstate = db:query(query, nrows)
```

说明：本方法是 send\_query 和 read\_result 组合的快捷方法。

这个方法需要自行检查 again，因为它只能调用一次 read\_result。参见下面的多结果集部分。

## 10. server\_ver

语法：

```
str = db:server_ver()
```

说明：返回服务器版本号字符串，如 “5.1.64”。

只有当成功连接到服务器时调用才会成功，否则会返回 nil。

## 11. set\_compact\_arrays

语法：

```
db:set_compact_arrays(boolean)
```

说明：设置是否使用 compact-arrays 结构。

### 12.1.4 多结果集返回示例

查询语句会产生多结果集返回的情况，这种情况下需要自行检查 query 和 read\_result 方法返回的“again”错误字符。

示例：

```
local cJSON = require "cjson"
local mysql = require "resty.mysql"

local db = mysql:new()
local ok, err, errcode, sqlstate = db:connect({
    host = "127.0.0.1",
    port = 3306,
    database = "world",
    user = "monty",
    password = "pass"})

if not ok then
    ngx.log(ngx.ERR, "failed to connect: ", err, ": ", errcode, " ", sqlstate)
    return ngx.exit(500)
end

res, err, errcode, sqlstate = db:query("select 1; select 2; select 3;")
if not res then
    ngx.log(ngx.ERR, "bad result #1: ", err, ": ", errcode, ": ", sqlstate, ".")
    return ngx.exit(500)
end

ngx.say("result #1: ", cJSON.encode(res))

local i = 2
while err == "again" do
    res, err, errcode, sqlstate = db:read_result()
    if not res then
        ngx.log(ngx.ERR, "bad result #", i, ": ", err, ": ", errcode, ": ", sqlstate, ".")
        return ngx.exit(500)
    end

    ngx.say("result #", i, ": ", cJSON.encode(res))
    i = i + 1
end

local ok, err = db:set_keepalive(10000, 50)
if not ok then
    ngx.log(ngx.ERR, "failed to set keepalive: ", err)
    ngx.exit(500)
end
```

这个代码段将生成下面的应答数据：

```
result #1: [{"1":"1"}]
result #2: [{"2":"2"}]
result #3: [{"3":"3"}]
```

## 12.1.5 其他注意事项

使用 lua-resty-mysql 库有若干需要注意的事项，下面将详细讲解。

### 1. SQL 文字引用（文法）

对 SQL 语句做好引用检查，以防止注入攻击。使用 ngx\_lua 提供的 ngx.quote\_sql\_str 函数引用参数值，例如：

```
local name = ngx.unescape_uri(ngx.var.arg_name)
local quoted_name = ngx.quote_sql_str(name)
local sql = "select * from users where name = " .. quoted_name
```

### 2. Debugging

需要使用 lua-cjson 库处理 query 方法返回的 JSON 数据，例如：

```
local cjson = require "cjson"
...
local res, err, errcode, sqlstate = db:query("select * from cats")
if res then
    print("res: ", cjson.encode(res))
end
```

### 3. 自动记录错误日志

默认地，当套接字发生错误时，ngx\_lua 模块会自动记录错误日志。如果你在自己的 Lua 代码中做了错误处理，推荐关闭自动日志功能。关闭 ngx\_lua 的 lua\_socket\_log\_errors 指令如下：

```
lua_socket_log_errors off;
```

## 12.1.6 限制

- 这个库不能在这些上下文使用：init\_by\_lua、set\_by\_lua、log\_by\_lua 和 header\_filter\_by\_lua。因为在这些上下文中 cosocket 对象是无效的。
- resty.mysql 不能在 Lua 模块级别被保存在 Lua 变量中，因为对象会被所在工作进程中的所有例程共享，当几个例程使用相同的 resty.mysql 对象时就会出现问題。只能在函数内部变量或 ngx.ctx 表里初始化 resty.mysql 对象。在这些地方，每一个请求都会放置自己的数据。

## 12.2 HttpDrizzleModule 访问方式

HttpDrizzleModule 提供了配置指令级的使用方式，相较于 lua-resty-mysql 使用更方便

一些，代码会更简洁。但是 HttpDrizzleModule 不支持存储过程的访问。

### 12.2.1 示例

下面这个示例演示了如何使用 libdrizzle 访问 MySQL。

首先编辑 nginx.conf，使用指令访问 MySQL。

```
http {
    ...

    upstream cluster {
        # simple round-robin
        drizzle_server 127.0.0.1:3306 dbname=test
            password=some_pass user=monty protocol=mysql;
        drizzle_server 127.0.0.1:1234 dbname=test2
            password=pass user=bob protocol=drizzle;
    }

    upstream backend {
        drizzle_server 127.0.0.1:3306 dbname=test
            password=some_pass user=monty protocol=mysql;
    }

    server {
        location /mysql {
            set $my_sql 'select * from cats';
            drizzle_query $my_sql;

            drizzle_pass backend;

            drizzle_connect_timeout    500ms; # default 60s
            drizzle_send_query_timeout 2s;    # default 60s
            drizzle_recv_cols_timeout  1s;    # default 60s
            drizzle_recv_rows_timeout  1s;    # default 60s
        }

        ...

        # for connection pool monitoring
        location /mysql-pool-status {
            allow 127.0.0.1;
            deny all;

            drizzle_status;
        }
    }
}
```

HttpDrizzleModule 访问 MySQL 非常简单，由配置指令和 API 两部分实现。使 upstream 模块和 libdrizzle 一起工作，可以提供一個非阻塞和流式的访问途径。

HttpDrizzleModule 模块本质上提供了一个非常有效果和灵活的方法访问 MySQL、Drizzle 或其他支持 Drizzle 或 MySQL 协议的 RDBMS，而且可以直接作为这些 RDBMS 端

点的 REST 接口。

这个接口不产生可读的输出，是一种叫作 RDS (Resty-DBD-Stream) 的二进制格式，所以解析输出需要其他的组件，如 rds-json-nginx-module、rds-csv-nginx-module、lua-rds-parser。

## 12.2.2 安装

ngx\_drizzle 模块默认是不打开的，需要在 configuring OpenResty 时使用 `--with-http_drizzle_module` 选项打开。

libdrizzle C 库不再是 OpenResty 包的一部分，所以需要自行下载，下载地址为 <https://launchpad.net/drizzle>。

### 1. 安装 libdrizzle 1.0

首先下载 libdrizzle 源码包：

```
wget http://agentzh.org/misc/nginx/drizzle7-2011.07.21.tar.gz
```

解压和编译：

```
tar xzvf drizzle7-2011.07.21.tar.gz
cd drizzle7-2011.07.21/
./configure --without-server
make libdrizzle-1.0
make install-libdrizzle-1.0
```

---

**注意：**make 的时候可能会失败，原因通常是缺少支撑的模块，根据错误提示把缺少的模块安装上即可，例如，缺少 g++ 编译器，则输入以下命令即可。

---

```
yum -y install gcc-g++
```

需要确保系统上已经安装指向 python2 的 python 界面，大多数最近的 Linux 商业版本默认已经是 python3，这在运行 libdrizzle-1.0 时会得到下面错误：

```
File "config/pandora-plugin", line 185
    print "Dependency loop detected with %s" % plugin['name']
                                         ^
SyntaxError: invalid syntax
make: *** [.plugin.scan] Error 1
```

把 python 指向 python2 就可以解决这个错误。

### 2. 安装 OpenResty

通过 yum 安装的 OpenResty 没有内置 HttpDrizzleModule，需要通过源码安装，可以直接覆盖 yum 安装，也可以安装新的 OpenResty。

安装支撑库：

```
yum install -y openssl-devel pcre-devel
```

下载源码包:

```
wget http://openresty.org/download/openresty-1.11.2.1.tar.gz
```

解压和安装:

```
tar zxvf openresty-1.11.2.1.tar.gz
cd openresty-1.11.2.1
./configure --with-http_drizzle_module --with-libdrizzle=/usr/local
gmake
gmake install
```

当把 libdrizzle-1.0 库安装到一个自定义的路径中时, 需要在 OpenResty 里指定前缀, 例如:

```
cd /path/to/nginx_openresty-VERSION/
./configure --with-libdrizzle=/path/to/drizzle --with-http_drizzle_module
```

### 12.2.3 技术点

HttpDrizzleModule 在使用中有两个技术点需要注意。

#### 1. Keepalive 连接池

HttpDrizzleModule 提供了一个内建的 MySQL 或 Drizzle TCP 工作进程级连接池。

下面是一个简单的配置:

```
upstream backend {
    drizzle_server 127.0.0.1:3306 dbname=test
    password=some_pass user=monty protocol=mysql;
    drizzle_keepalive max=100 mode=single overflow=reject;
}
```

连接池使用一个简单的 LIFO 算法分配空闲连接。最近使用的连接会在下次首先复用。在连接池满载的情况下, 新的空闲连接将覆盖旧的空闲连接。

#### 2. Last Insert ID

如果要获取 LAST\_INSERT\_ID, 当执行一个 SQL 插入查询的时候, ngx\_drizzle 会自动返回这个值。

```
location /test {
    echo_location /mysql "drop table if exists foo";
    echo;
    echo_location /mysql "create table foo (id serial not null, primary key (id),
val real);";
    echo;
    echo_location /mysql "insert into foo (val) values (3.1415926);";
    echo;
    echo_location /mysql "select * from foo;";
    echo;
}

location /mysql {
```

```

drizzle_pass backend;
drizzle_module_header off;
drizzle_query $query_string;
rds_json on;
}

```

请求 /test 会得到下面输出:

```

{"errcode":0}
{"errcode":0}
{"errcode":0,"insert_id":1,"affected_rows":1}
[{"id":1,"val":3.1415926}]

```

## 12.2.4 配置指令

HttpDrizzleModule 访问方式主要涉及以下配置指令。

### 1. drizzle\_server

语法:

```

drizzle_server <host> user=<user> password=<pass> dbname=<database>
drizzle_server <host>:<port> user=<user> password=<pass> dbname=<database>
protocol=<protocol> charset=<charset>

```

默认: no。

上下文: upstream。

说明: 配置 MySQL 服务器名字和参数。名字可以是域名、地址, 可以配合一个可选的端口号 (默认 3306), 如果域名可以解析为多个地址, 那么多个地址都将会使用。

配置 drizzle\_server, 支持下面可选参数。

- user=<user>: MySQL/Drizzle 用户名, 用于登录。
- password=<pass>: 登录密码。如果有特殊符号, 需要使用双括号或单括号括起来, 如密码中有 #:

```

drizzle_server 127.0.0.1:3306 user=monty "password=a b#1"
      dbname=test protocol=mysql;

```

- dbname=<database>: 指定默认 MySQL 数据库。
- protocol=<protocol>: 指定使用的协议, 即 Drizzle 或 MySQL。默认是 Drizzle
- charset=<charset>: 编码格式。如果默认的格式正是你需要使用的格式, 则不要重新使用本参数指定, 否则会造成运行期的性能占用。例如:

```

drizzle_server foo.bar.com:3306 user=monty password=some_pass
      dbname=test protocol=mysql
      charset=utf8;

```

### 2. drizzle\_keepalive

语法:

```

drizzle_keepalive max=<size> mode=<mode>

```

默认:

```
drizzle_keepalive max=0 mode=single
```

上下文: upstream。

说明: 配置 MySQL/Drizzle 连接池, 支持下面可选参数。

- **max=<num>**: 指定当前 upstream 块连接池最大能力。值必须是非 0 值, 如果是 0, 则表示禁用连接池, 默认是 0。
- **mode=<mode>**: 支持 single 和 multi。single 模式表示连接池不区分当前块中多个 Drizzle 服务器, multi 意味着连接池将按服务器名字和端口进行连接池匹配和调度。默认是 single。
- **overflow=<action>**: 指定连接池满的时候如何处理新来的请求, 支持 reject 或 ignore。reject 情况下将拒绝新的请求, 返回 503 错误。ignore 情况下将继续创建新的连接。

### 3. drizzle\_query

语法:

```
drizzle_query <sql>
```

默认: no。

上下文: http、server、location、location if。

说明: 向 Drizzle/MySQL 发送查询命令。

支持 Nginx 变量, 但是要小心 SQL 注入攻击, 可以使用 set\_quote\_sql\_str 指令, 例如:

```
location /cat {
    set_unescape_uri $name $arg_name;
    set_quote_sql_str $quoted_name $name;

    drizzle_query "select * from cats where name = $quoted_name";
    drizzle_pass my_backend;
}
```

### 4. drizzle\_pass

语法:

```
drizzle_pass <remote>
```

默认: no。

上下文: location、location if。

阶段: content。

说明: 指定 Drizzle 或 MySQL upstream 名字, <remote> 可以是包含 drizzle\_server 的任意 upstream。

在 <remote> 中使用 Nginx 变量, 即可实现动态端点路由, 例如:



```
upstream moon { drizzle_server ...; }
```

```
server {
    location /cat {
        set $backend 'moon';

        drizzle_query ...;
        drizzle_pass $backend;
    }
}
```

### 5. drizzle\_connect\_timeout

语法:

```
drizzle_connect_time <time>
```

默认:

```
drizzle_connect_time 60s
```

上下文: http、server、location、location if。

说明: 指定连接到远程 Drizzle 或 MySQL 服务器的连接超时值。<time> 值可以是整数, 单位是秒 (s)、毫秒 (ms)、分钟 (m)。默认是 60 秒。

### 6. drizzle\_send\_query\_timeout

语法:

```
drizzle_send_query_timeout <time>
```

默认:

```
drizzle_send_query_timeout 60s
```

上下文: http、server、location、location if。

说明: 指定发送到远程服务器查询指令的超时值。单位和默认值同 drizzle\_connect\_timeout 指令。

### 7. drizzle\_recv\_cols\_timeout

语法:

```
drizzle_recv_cols_timeout <time>
```

默认:

```
drizzle_recv_cols_timeout 60s
```

上下文: http、server、location、location if。

说明: 指定从服务器接收列结果集的超时值。

### 8. drizzle\_recv\_rows\_timeout

语法:

```
drizzle_recv_rows_timeout <time>
```

默认:

```
drizzle_recv_rows_timeout 60s
```

上下文: http、server、location、location if。

说明: 指定从服务器接收行结果集数据的超时值。

## 9. drizzle\_buffer\_size

语法:

```
drizzle_buffer_size <size>
```

默认:

```
drizzle_buffer_size 4k/8k
```

上下文: http、server、location、location if。

说明: 指定 drizzle 输出的缓冲区, 默认页是 4k/8k。

## 10. drizzle\_module\_header

语法:

```
drizzle_module_header on|off
```

默认:

```
drizzle_module_header on
```

上下文: http、server、location、location。

说明: 控制是否在应答中加入 drizzle 头。默认是 on。

## 11. drizzle\_status

语法:

```
drizzle_status
```

默认: no。

上下文: location、location if。

阶段: content。

说明: 打开时, Nginx 的 location 为所有的 drizzle upstream 中的 servers 输出状态报告。

输出样例如下:

```
worker process: 15231
```

```
upstream backend
```

```
  active connections: 0
```

```
  connection pool capacity: 10
```

```
  overflow: reject
```

```
  cached connection queue: 0
```

```

free'd connection queue: 10
cached connection successfully used count:
free'd connection successfully used count: 3 0 0 0 0 0 0 0 0
servers: 1
peers: 1

upstream backend2
    active connections: 0
    connection pool capacity: 0
    servers: 1
    peers: 1

```

如果有多个工作者进程，这个报告不是全局的统计，就是每个工作进程中的 upstream。

### 12.2.5 变量

drizzle-nginx-module 创建了新的 Nginx 变量：

```
$drizzle_thread_id
```

当前 SQL 查询超时，这个变量返回一个对应 MySQL 或 Drizzle 线程 ID 的字符串，用于将来发送 SQL kill 命令以取消超时的查询。

例如：

```

drizzle_connect_timeout 1s;
drizzle_send_query_timeout 2s;
drizzle_recv_cols_timeout 1s;
drizzle_recv_rows_timeout 1s;

location /query {
    drizzle_query 'select sleep(10)';
    drizzle_pass my_backend;
    rds_json on;

    more_set_headers -s 504 'X-MySQL-Tid: $drizzle_thread_id';
}

location /kill {
    drizzle_query "kill query $arg_tid";
    drizzle_pass my_backend;
    rds_json on;
}

location /main {
    content_by_lua '
        local res = ngx.location.capture("/query")
        if res.status == ngx.HTTP_OK then
            local tid = res.header["X-MySQL-Tid"]
            if tid and tid ~= "" then
                ngx.location.capture("/kill", { args = {tid = tid} })
            end
            return ngx.HTTP_INTERNAL_SERVER_ERROR;
        end
    '
}

```

```

        end
        ngx.print(res.body)
    }

```

在使用 `headers_more_nginx_module`、`lua_nginx_module` 和 `rds_json_nginx_module` 的时候，若 SQL 查询超时，我们可以立即明确取消该查询操作。这里有一个陷阱，你需要在创建 Nginx 时按以下顺序添加这些模块：

- `lua_nginx_module`。
- `headers_more_nginx_module`。
- `rds_json_nginx_module`。

这会决定它们的输出过滤器工作在倒序情况下。例如，第一个转换 RDS 到 JSON，另一个添加自定义 X-Mysql-Tidcustom 头，最后一个使用 Lua 模块捕获整个应答，推荐使用 OpenResty，它自动按正确顺序绑定模块。

## 12.2.6 输出格式

本模块输出二进制查询结果，多数 Nginx 数据模块，如 `ngx_postgres`，都遵守 RDS 格式。如果你是一个 Web APP 开发者，可能对这些更感兴趣：

- 使用 `rds_json_nginx_module` 获得 JSON 输出。
- 使用 `rds_csv_nginx_module` 获得 Comma-Separated-Value (CSV) 输出。
- 使用 `lua-rds-parser` 将 RDS 解析成 Lua 数据格式。

HTTP 应答头部分总是返回 200。Content-Type 必须设置为 `application/x-resty-dbd-stream`。驱动生成应答头部时总是设置为 X-Resty-DBD，例如：

```
X-Resty-DBD-Module: drizzle 0.1.0
```

X-Resty-DBD-Module 是可选的，下面是 HTTP 应答包体。

### 1. RDS 头域

RDS 头由下面域组成：

- `uint8_t`：字节序类型 (1 表示大端模式，其他表示小端模式)。
- `uint32_t`：格式版本 (v1.2.3 表示 1002003 这个数值)。
- `uint8_t`：结果类型 (0 表示正常 SQL 结果类型，现在仅这个固定值)。
- `uint16_t`：标准错误码。
- `uint16_t`：驱动指定错误码。
- `uint16_t`：驱动指定错误字符串长度。
- `*u_char **`：驱动指定错误字符串。
- `uint64_t`：数据库受影响行数。
- `uint64_t`：插入 ID( 如果没有，值为 0)。
- `uint16_t`：列数。

## 2. RDS 包体

若头域的列数域是 0，则整个 RDS 包体部分将被忽略。

RDS 包体由两部分组成：Columns 和 Rows。

### (1) Columns

列部分存放列数据，数量由头部列数部分指定。

每一列由下面域组成：

- uint16\_t: 非零值表示标准列类型码和结束符。
- uint16\_t: 驱动指定的列类型码。
- uint16\_t: 列名长度。
- \*u\_char \*\*: 列名数据。

### (2) Rows

行由 0 或更多原始数据组成，使用一个 8bit 的 0 表示结束。

每一个原始数据由 Row Flag 和一个可选的 Fields Data 部分组成。

#### 1) Row Flag。

- uint8\_t: 有效行 (1 表示有效，0 表示结果集列表结束，没有更多行了)。

#### 2) Fields Data: 由 0 或更多的域数据组成。数量由头部的列数域指定。

- uint32\_t: 域长度 (-1 表示 null)。
- \*u\_char \*\*: 文本定义的域数据，如果域长度为 -1，本域为空串。

## 3. Status Code

如果 MySQL 查询错误码不是 OK，则返回一个 500 的错误页（除了表不存在错误，这个返回 410 错误）。

## 12.3 HttpDrizzleModule 完整示例

下面的 nginx.conf 定义了两个 location，用于访问 MySQL，相当于封装了两个 RESTful API 供内部调用，如果监听的 IP 是局域网的，则可以只为本网提供服务，防止外来访问，同时可以使用 access 进行访问控制：

```
#nginx.conf
user root;
worker_processes 4;
worker_rlimit_nofile 100000;

error_log logs/error.log;

pid logs/nginx.pid;

events{
    use epoll;
```

```

    worker_connections 10000;
}

http{
    include      mime.types;
    default_type text/html;

    access_log    off;
    server_tokens off;

    sendfile      on;
    tcp_nopush    on;
    tcp_nodelay    on;
    open_file_cache max=10240 inactive=60s;
    open_file_cache_valid 80s;
    open_file_cache_min_uses 1;
    lua_shared_dict gkey 50m;

    keepalive_timeout 0;
    chunked_transfer_encoding off;

    upstream bk_mysql {
        drizzle_server 10.12.10.15:3306 protocol=mysql dbname=test user=mgr
        password="233wk%"
        drizzle_keepalive max=300 overflow=reject mode=single;
    }

    upstream bk_redis{
        server 10.12.10.5:69;
        keepalive 1000;
    }

    server{
        listen 9000 default so keepalive=on;
        server_name ourtestdrizzle;
        charset utf-8;

        location /mysql {
            include /usr/local/ip_limit.conf;
            drizzle_pass bk_mysql;
            drizzle_query $request_body;
        }

        location /exec_sql {
            include /usr/local/ip_limit.conf;
            content_by_lua '
                local sql=ngx.unescape_uri(ngx.var.arg_sql)
                local resp = ngx.location.capture("/mysql", {method = ngx.HTTP_POST,
                body = sql})
                if resp.status ~= ngx.HTTP_OK or not resp.body then
                    ngx.exit(resp.status)
                end
                ngx.print(resp.body)
            '
        }
    }
}

```

```
}
}
```

在 server 里增加一个 location，对外提供 rest 服务，服务调用上面的内部子请求访问数据库，执行用户的请求：

```
#####add gateway #####
    location /gateway/add {
        lua_need_request_body on;
        client_max_body_size 50k;
        client_body_buffer_size 50k;
        content_by_lua '

local vdata={}
local json_data = ngx.var.request_body
ngx.log(ngx.ERR, json_data)
local cJSON = require "cjson"

vdata = cJSON.decode(json_data)
local gwId = vdata["gwId"]
local gwName = vdata["gwName"]
local model = vdata["model"]
local interval = vdata["interval"]
local isAlarm = vdata["isAlarm"]
local isOnline = vdata["isOnline"]
local firewall = vdata["firewall"]
local region = vdata["region"]
local gps = vdata["gps"]
local userId = vdata["userId"]

local msg="{\034msg\034:\034failure\034}"
if gwId == nil then
    msg="{\034msg\034:\034failure\034,\034desc\034:\034gateway id is null!\034}"
    ngx.print(msg)
    ngx.exit(ngx.HTTP_OK)
end
if userId == nil then
    msg="{\034msg\034:\034failure\034,\034desc\034:\034user id is null!\034}"
    ngx.print(msg)
    ngx.exit(ngx.HTTP_OK)
end
if model == nil then
    msg="{\034msg\034:\034failure\034,\034desc\034:\034model field is null!\034}"
    ngx.print(msg)
    ngx.exit(ngx.HTTP_OK)
end

local sql="SELECT sid,userId from gateway WHERE sid=\'" .. gwId .. "\'"
local resp= ngx.location.capture("/exec_sql?sql=" .. ngx.escape_uri(sql), {method
    = ngx.HTTP_GET})

msg="{\034msg\034:\034failure\034,\034desc\034:\034server error!\034}"

if resp.status ~= ngx.HTTP_OK or not resp.body then
    ngx.print(msg)
```

```

    ngx.exit(501)
end

local parser = require "rds.parser"
local res, err = parser.parse(resp.body)
if res == nil then
    msg="{\034msg\034:\034failure\034,\034desc\034:\034server error!\034}"
    ngx.print(msg)
    ngx.exit(501)
end

local rows = res.resultset

if #rows > 0 then
    msg="{\034msg\034:\034failure\034,\034desc\034:\034gateway already exist!\034}"
    ngx.print(msg)
    ngx.exit(ngx.HTTP_OK)
end

if gwName == nil then
    gwName = ""
end
if isAlarm == nil then
    isAlarm = 0
end
if gps == nil then
    gps=""
end
if interval == nil then
    interval = 10000
end
if firmware == nil then
    firmware = ""
end
if region == nil then
    region = ""
end
if model == nil then
    model = ""
end
if vender == nil then
    vender = ""
end

sql="INSERT INTO gateway (sid, userId, gwName, isAlarm, intervaltime, gps,
    firmware, model, vender, region) VALUES ('\'' .. gwId .. '\'' .. userId ..
    '\'' .. gwName .. '\'' .. isAlarm .. '\'' .. interval .. '\'' ..
    gps .. '\'' .. firmware .. '\'' .. model .. '\'' .. vender ..
    '\'' .. region .. '\'')"
for i = 1, 4, 1 do
    resp= ngx.location.capture("/exec_sql?sql=" .. ngx.escape_uri(sql), {method
        = ngx.HTTP_GET})
    if resp.status == ngx.HTTP_OK and resp.body then
        break
    end
end

```



```

end
ngx.sleep(0.5)
end
if resp.status ~= ngx.HTTP_OK or not resp.body then
    --ngx.say("failed to query mysql")
    msg="{\034msg\034:\034failure\034,\034desc\034:\034failed to insert
gateway!\034}"
    ngx.print(msg)
    ngx.exit(501)
end
local parser = require "rds.parser"
local res, err = parser.parse(resp.body)
if res == nil then
    msg="{\034msg\034:\034failure\034,\034desc\034:\034failed to parse
rds!\034}"
    ngx.print(msg)
    ngx.exit(501)
end

local rows = res.affected_rows
if rows == nil then
    rows=0
end

if rows>0 then
    msg="{\034msg\034:\034successfully\034,\034desc\034:\034OK!\034}"
    ngx.print(msg)
    ngx.exit(ngx.HTTP_OK)
else
    msg="{\034msg\034:\034failure\034,\034desc\034:\034failed to add
gateway!\034}"
    ngx.print(msg)
    ngx.exit(501)
end
end

```

下面是一个完整处理 RDS 的例子:

```

location /user/getGateways {
    lua_need_request_body on;
    client_max_body_size 50k;
    client_body_buffer_size 50k;
    #access_by_lua_file access.lua;
    content_by_lua '
local userId=ngx.var.arg_userId
local resultStr
local vdata={}
local gateways={}
local sensors={}
local cJSON = require "cjson"

if userId == nil then
    msg="{\034msg\034:\034failure\034,\034desc\034:\034user id is null!\034}"
    ngx.print(msg)
    ngx.exit(ngx.HTTP_OK)
end

```

```

local sql="SELECT sid,gwName,model,vender,isAlarm,state,firmware,intervaltime,
        region,gps FROM gateway WHERE userid='" .. userId ..'"
local resp= ngx.location.capture("/exec_sql?sql=" .. ngx.escape_uri(sql), {method
        = ngx.HTTP_GET})
msg="{\034msg\034:\034failure\034,\034desc\034:\034server error!\034}"

if resp.status ~= ngx.HTTP_OK or not resp.body then
    ngx.print(msg)
    ngx.exit(501)
end

local parser = require "rds.parser"
local res, err = parser.parse(resp.body)
if res == nil then
    ngx.exit(501)
end

local affected_rows = 0
local rows = res.resultset
local gwId
local gwName
local model
local vender
local isAlarm
local isOnline
local firmware
local intervertime
local region
local gps
local i,row,col,val
local j,jrow,jcol,jval
local jparser
local jres,jerr, jresp, jrows
local sensorId
local sensorName
local sensorType
local sensorUnit
local sensorOnline
local sensorAlarm
local sensorRegion
local sensorGps
if rows == nil then
    ngx.say("no gateway.")
    ngx.exit(403)
end

for i, row in ipairs(rows) do
    for col, val in pairs(row) do
        if col=="sid" then
            gwId = val
        elseif col=="gwName" then
            gwName = val
        elseif col=="model" then
            model = val

```

```

elseif col=="vender" then
    vender = val
elseif col=="isAlarm" then
    isAlarm = val
elseif col=="isOnline" then
    isOnline=val
elseif col=="firmware" then
    firmware=val
elseif col=="intervaltime" then
    intervaltime=val
elseif col=="region" then
    region=val
elseif col=="gps" then
    gps=val
end
end

```

```

sql="SELECT sensorId,name,type,unit,isAlarm,isOnline,region,gps FROM sensor
WHERE gwId=\'\' .. gwId .. \'\'

```

```

jresp= ngx.location.capture("/exec_sql?sql=" .. ngx.escape_uri(sql), {method =
ngx.HTTP_GET})

```

```

if jresp.status ~= ngx.HTTP_OK or not jresp.body then
    break
end

```

```

jparser = require "rds.parser"
jres, jerr = jparser.parse(jresp.body)
if jres == nil then
    break
end

```

```

jrows = jres.resultset
for j, jrow in ipairs(jrows) do
    for jcol, jval in pairs(jrow) do
        if jcol=="sensorId" then
            sensorId=jval
        elseif jcol=="name" then
            sensorName=jval
        elseif jcol=="type" then
            sensorType=jval
        elseif jcol=="isOnline" then
            sensorOnline=jval
        elseif jcol=="isAlarm" then
            sensorAlarm=jval
        elseif jcol=="region" then
            sensorRegion=jval
        elseif jcol=="gps" then
            sensorGps=jval
        end
    end
end

```

```

vdata["sensorId"]=sensorId
vdata["name"]=sensorName
vdata["type"]=sensorType
vdata["unit"]=sensorUnit
vdata["isOnline"]=sensorOnline

```

```

        vdata["isAlarm"]=sensorAlarm
        vdata["region"]=sensorRegion
        vdata["gps"]=sensorGps
        table.insert(sensors, vdata)
        vdata={}
    end

    vdata["gwId"]=gwId
    vdata["gwName"]=gwName
    vdata["isAlarm"]=isAlarm
    vdata["isOnline"]=isOnline
    vdata["firmware"]=firmware
    vdata["interval"]=intervaltime
    vdata["region"]=region
    vdata["gps"]=gps
    vdata["sensors"]=sensors

    table.insert(gateways, vdata)
    vdata={}
end

vdata["gateways"]=gateways
local resultStr=cjson.encode(vdata)
ngx.say(resultStr)
ngx.exit(200)

';
}

```

## 12.4 小结

本章介绍了 HttpDrizzleModule 和 lua-resty-mysql 两种 MySQL 访问方式。HttpDrizzleModule 结合 upstream 模块，访问速度快，使用简单，但是这种方式不支持存储过程。lua-resty-mysql 是 OpenResty 团队开发的组件，所占内存更小，使用更灵活，支持存储过程，但编码工作量大。我们应该根据具体工程的规模 and 实际使用情况选择 MySQL 访问方式。

## Memcached 操作

访问 Memcached 有两种方式：mem-nginx-module 访问方式和 lua-resty-memcached 访问方式。lua-resty-memcached 是由 OpenResty 实现的，mem-nginx-module 也被包含在 OpenResty 内。使用 OpenResty 可以直接使用这两种方式访问 Memcached，自行安装 Nginx 的需要加入这两个模块才可以使用。

### 13.1 mem-nginx-module 访问方式

ngx\_memc 是标准 Memcached 模块的一个扩展版本，支持 set、add、delete 和其他 Memcached 命令。ngx\_memc 实现了 Memcached ascii 协议，可以使用本模块快速实现 Memcached 的 REST 接口，可作为其他模块的子请求提供服务，或对外服务。如果使用本模块缓存 location 应答，最好和 srcache-nginx-module 配合实现。

如果在 lua-nginx-module 中使用，推荐使用 lua-resty-memcached 库替代，因为其更灵活，更节省内存。

本模块不随 Nginx 源码发行，在 Nginx 中需要单独安装。

#### 13.1.1 概述

这里通过 6 个示例演示 ngx\_memc 的使用方法。ngx\_memc 通过配置指令实现，通常通过子请求向其他模块提供服务。

##### 1. 最简单访问示例

下面是一个最简单的示例：

```

# GET /foo?key=dog
#
# POST /foo?key=cat
# Cat's value...
#
# PUT /foo?key=bird
# Bird's value...
#
# DELETE /foo?key=Tiger
location /foo {
    set $memc_key $arg_key;

    # $memc_cmd defaults to get for GET,
    # add for POST, set for PUT, and
    # delete for the DELETE request method.

    memc_pass 127.0.0.1:11211;
}

```

上面的例子配置了一个名为 /foo 的 location，我们可以使用如下的 URL 访问这个 location：

1. GET /foo?key=dog
2. POST /foo?key=cat
3. PUT /foo?key=bird
4. DELETE /foo?key=Tiger

key 是我们传入的 key。memc\_cmd 默认使用将请求的 GET 方法对应为 get 操作，将 POST 操作对应为 add，将 PUT 对应为 set，将 DELETE 对应为 delete。

## 2. 带 cmd 操作和 key 的示例

```

# GET /bar?cmd=get&key=cat
#
# POST /bar?cmd=set&key=dog
# My value for the "dog" key...
#
# DELETE /bar?cmd=delete&key=dog
# GET /bar?cmd=delete&key=dog
location /bar {
    set $memc_cmd $arg_cmd;
    set $memc_key $arg_key;
    set $memc_flags $arg_flags; # defaults to 0
    set $memc_exptime $arg_exptime; # defaults to 0

    memc_pass 127.0.0.1:11211;
}

```

这个示例定义了一个名叫 /bar 的 location，访问这个 location 时需要提供两个参数，一个是 cmd 操作，另一个是 key 值。另外两个参数 flags 和 exptime 因为没有使用，所以默认值是 0。

访问的方法如下：

1. GET /bar?cmd=get&key=cat
2. POST /bar?cmd=set&key=dog
3. DELETE /bar?cmd=delete&key=dog
4. GET /bar?cmd=delete&key=dog

### 3. 使用全部参数的示例

```
# GET /bar?cmd=get&key=cat
# GET /bar?cmd=set&key=dog&val=animal&flags=1234&exptime=2
# GET /bar?cmd=delete&key=dog
# GET /bar?cmd=flush_all
location /bar {
    set $memc_cmd $arg_cmd;
    set $memc_key $arg_key;
    set $memc_value $arg_val;
    set $memc_flags $arg_flags; # defaults to 0
    set $memc_exptime $arg_exptime; # defaults to 0

    memc_cmds_allowed get set add delete flush_all;

    memc_pass 127.0.0.1:11211;
}
```

这个示例使用了更多的参数和配置，参数多了 val，同时配置了模块允许的操作：

```
memc_cmds_allowed get set add delete flush_all;
```

可以访问 URL 如下：

- 1.GET /bar?cmd=get&key=cat
- 2.GET /bar?cmd=set&key=dog&val=animal&flags=1234&exptime=2
- 3.GET /bar?cmd=delete&key=dog
- 4.GET /bar?cmd=flush\_all

### 4. stats 操作示例

```
http {
    ...
    upstream backend {
        server 127.0.0.1:11984;
        server 127.0.0.1:11985;
    }
    server {
        location /stats {
            set $memc_cmd stats;
            memc_pass backend;
        }
        ...
    }
    ...
}
```

这个示例定义了一个 /stats 的 location，访问这个 URL 会得到 Memcached 的 stats 信息。

## 5. 标志读进 Last-Modified 示例

```
# read the memcached flags into the Last-Modified header
# to respond 304 to conditional GET
location /memc {
    set $memc_key $arg_key;

    memc_pass 127.0.0.1:11984;

    memc_flags_to_last_modified on;
}
```

这个示例中将 Memcached 的标志读进 Last-Modified 头域，用于响应 304 的 GET 操作。

## 6. 访问 UNIX 域套接字服务示例

```
location /memc {
    set $memc_key foo;
    set $memc_cmd get;

    # access the unix domain socket listend by memcached
    memc_pass unix:/tmp/memcached.sock;
}
```

这个示例访问 Memcached 监听的 UNIX 域套接字。

### 13.1.2 命令

本节介绍 Memcached 命令在 ngx\_memc 中的实现。因为 ngx\_memc 模块是通过 upstream 机制实现的，所以所有命令都是在 nginx.conf 中通过配置命令实现的，参数通过变量传递。

与其他的库不同，其他的库对应的是函数和 API，ngx\_memc 对应的是命令在配置文件中的实现。

#### 1. get \$memc\_key

功能：获取 key 的值。

```
location /foo {
    set $memc_cmd 'get';
    set $memc_key $arg_key;
    memc_pass 127.0.0.1:11211;
    add_header X-Memc-Flags $memc_flags;
}
```

如果 key 找到了，将在应答包体中返回 200 ok，否则返回 404 (not found)。flags 的数值型值放在 \$mem\_flags 变量中，经常使用 add\_header 指令在应答头中放进这条信息。在 ERROR、CLIENT\_ERROR、SERVER\_ERROR 情况下返回 502 错误。

数据存储在包体里，可以直接使用包体变量取出值。



在 Lua 代码中，使用子请求访问 REST 风格的接口，以子请求方式访问，直接通过包体返回对象读取数值：

```
...
local resp
resp = ngx.location.capture("/foo?key=" .. val)
if resp.status == ngx.HTTP_OK and resp.body then
    print(resp.body)
end
...
```

## 2. set \$memc\_key \$memc\_flags \$memc\_exptime \$memc\_value

功能：使用请求包体作为 Memcached 值，避免设置 \$memc\_value 变量。

```
# POST /foo
# my value...
location /foo {
    set $memc_cmd 'set';
    set $memc_key 'my_key';
    set $memc_flags 12345;
    set $memc_exptime 24;

    memc_pass 127.0.0.1:11211;
}
```

或使用 \$memc\_value 保存值：

```
location /foo {
    set $memc_cmd 'set';
    set $memc_key 'my_key';
    set $memc_flags 12345;
    set $memc_exptime 24;
    set $memc_value 'my_value';

    memc_pass 127.0.0.1:11211;
}
```

如果服务端返回 STORED 状态，函数返回 201 状态码。其他状态码为 404 (not found)、502 (ERROR、CLIENT\_ERROR、SERVER\_ERROR)。

除了 404-not found 外，原始的 Memcached 应答在包体内返回。

## 3. add \$memc\_key \$memc\_flags \$memc\_exptime \$memc\_value

功能：和 set 命令一样。

## 4. replace \$memc\_key \$memc\_flags \$memc\_exptime \$memc\_value

功能：和 set 命令一样。

## 5. append \$memc\_key \$memc\_flags \$memc\_exptime \$memc\_value

功能：和 set 命令一样。

---

注意：Memcached 1.2.2 版本不支持 append 命令和 prepend 命令。Memcached 1.2.4 以上版本才支持这两个命令。

---

## 6. prepend \$memc\_key \$memc\_flags \$memc\_exptime \$memc\_value

功能：和 append 命令一样。

## 7. delete \$memc\_key

功能：删除 key 的值对数据。

```
location /foo
    set $memc_cmd delete;
    set $memc_key my_key;
```

```
    memc_pass 127.0.0.1:11211;
```

```
}
```

删除成功返回 200，其他错误为 404 (not found)、502 (ERROR、CLIENT\_ERROR、SERVER\_ERROR)。

## 8. delete \$memc\_key \$memc\_exptime

功能：和 delete \$memc\_key 命令很相似，除了多接受一个可选的过期时间参数，保存在变量 \$memc\_exptime 里。这个命令在最新的 Memcached 1.4.4 版本里不再有效。

## 9. incr \$memc\_key \$memc\_value

功能：将 \$memc\_key 的值增长 \$memc\_value。

```
location /foo {
    set $memc_key my_key;
    set $memc_value 2;
    memc_pass 127.0.0.1:11211;
}
```

上面的例子中，每次访问 /foo 都会引起 my\_key 增长 2。

执行成功返回 200 OK，没找到的情况下返回 404 (not found)，ERROR、CLIENT\_ERROR、SERVER\_ERROR 情况下返回 502。

## 10. decr \$memc\_key \$memc\_value

功能：同 incr \$memc\_key \$memc\_value。

## 11. flush\_all

功能：使 Memcached 中的所有 key 过期。

```
location /foo {
    set $memc_cmd flush_all;
    memc_pass 127.0.0.1:11211;
}
```

## 12. flush\_all \$memc\_exptime

功能：和 flush\_all 不同的是通过 \$memc\_exptime 接收一个过期时间。

## 13. stats

功能：获取 Memcached 输出统计信息和设置。

```
location /foo {
    set $memc_cmd stats;
    memc_pass 127.0.0.1:11211;
}
```

查询成功返回 200，否则返回 502 (ERROR、CLIENT\_ERROR、SERVER\_ERROR)。统计信息在包体里。

## 14. 版本

功能：查询服务器版本。

```
location /foo {
    set $memc_cmd version;
    memc_pass 127.0.0.1:11211;
}
```

成功返回 200，否则返回 502 (ERROR、CLIENT\_ERROR、SERVER\_ERROR)。输出在应答的包体里。

### 13.1.3 指令

指令是 ngx\_memc 在 nginx.conf 中使用的配置指令，配置指令控制模块的参数和行为，模块按照配置执行操作。

#### 1. memc\_pass

语法：

```
memc_pass <memcached server IP address>:<memcached server port>
memc_pass <memcached server hostname>:<memcached server port>
memc_pass <upstream_backend_name>
memc_pass unix:<path_to_unix_domain_socket>
```

默认：none。

上下文：http、server、location、if。

阶段：content。

说明：指定 Memcached 端点。

#### 2. memc\_cmds\_allowed

语法：

```
memc_cmds_allowed <cmd>...
```

默认：none。

上下文：http、server、location、if。

说明：允许使用 Memcached 命令列表功能。默认地，所有支持的命令都是可访问的，例如：

```
location /foo {
```

```

set $memc_cmd $arg_cmd;
set $memc_key $arg_key;
set $memc_value $arg_val;

memc_pass 127.0.0.1:11211;

memc_cmds_allowed get;
}

```

### 3. memc\_flags\_to\_last\_modified

语法:

```
memc_flags_to_last_modified on|off
```

默认: off。

上下文: http、server、location、if。

说明: 读取 Last-Modified 时间。GET 操作中, Nginx 将返回 304 (Not Modified) 应答以节省带宽。

### 4. memc\_connect\_timeout

语法:

```
memc_connect_timeout <time>
```

默认: 60s。

上下文: http、server、location。

说明: 连接超时值, 默认单位是秒。

### 5. memc\_send\_timeout

语法:

```
memc_send_timeout <time>
```

默认: 60s。

上下文: http、server、location。

说明: 发送的超时值, 默认单位是秒。

### 6. memc\_read\_timeout

语法:

```
memc_read_timeout <time>
```

默认: 60s。

上下文: http、server、location。

说明: 读取超时值, 默认单位是秒。

### 7. memc\_buffer\_size

语法:

```
memc_buffer_size <size>
```

默认: 4K/8K。

上下文: http、server、location。

说明: 缓冲区大小。默认是页尺寸, 可以是 4KB 或 8KB。

## 8. memc\_ignore\_client\_abort

语法:

```
memc_ignore_client_abort on|off
```

默认: off。

上下文: location。

说明: 是否到 Memcache 的连接不等待应答就关闭。

## 13.1.4 安装

使用 OpenResty 时, 不需要单独安装, 如果自己使用 Nginx 源码进行安装, 则需要参照下面的步骤安装模块。

```
wget 'http://nginx.org/download/nginx-1.9.15.tar.gz'
tar -xzvf nginx-1.9.15.tar.gz
cd nginx-1.9.15/
```

```
# Here we assume you would install you nginx under /opt/nginx/.
./configure --prefix=/opt/nginx \
  --add-module=/path/to/memc-nginx-module
```

```
make -j2
make install
```

下载最新版本的 mem-nginx-module 模块文件。也可以在 Nginx 中使用动态模块, 在上面的 ./configure 命令中使用 --add-dynamic-module=PATH 代替 --add-module=PATH。然后在 nginx.conf 通过 load\_module 指令明确装载模块, 例如:

```
load_module /path/to/modules/ngx_http_memc_module.so;
```

## 13.1.5 说明

在使用 mem-nginx-module 时, 也有两个需要注意的事项。

### 1. 连接池

使用 mem-nginx-module 时, 推荐使用 HttpUpstreamKeepaliveModule 实现后端 Memcached 服务器连接缓存, 使用连接池会显著提升访问的速度。下面是一个样例配置:

```
http {
    upstream backend {
        server 127.0.0.1:11211;
```

```

# a pool with at most 1024 connections
# and do not distinguish the servers:
keepalive 1024;
}

server {
    ...
    location /memc {
        set $memc_cmd get;
        set $memc_key $arg_key;
        memc_pass backend;
    }
}
}

```

## 2. 支持的 Memcached 命令

mem-nginx-module 支持的 Memcached 命令为 set、add、replace、prepend、append。配置这些命令时，支持下面的变量作为参数：

- \$memc\_cmd 作为命令。
- \$memc\_key 作为 key。
- \$memc\_exptime 作为过期时间（或延迟，默认为 0）。
- \$memc\_flags 作为标志（默认为 0），按照这个规则生成请求命令。
- 如果 \$memc\_value 没有定义，除了 incr 和 decr 命令外，请求包体将作为 \$memc\_value。如果 \$memc\_value 定义了一个空字符串（""），那么这个空字符串仍然有效会被使用。

### 13.1.6 示例

下面是一个比较完整的例子，演示了 Memcached 的常用操作。

#### 1. 配置 nginx.conf

```

worker_processes 1;
events {
    worker_connections 1024;
}
http {
    include mime.types;
    default_type application/octet-stream;
    sendfile on;
    keepalive_timeout 65;
    upstream memcached {
        server 127.0.0.1:11211;
    }
    server {
        listen 80;
        server_name localhost;
        root html;
    }
}

```

```

index index.html index.htm;
location = /memcached-status {
    set $memc_cmd stats;
    memc_pass memcached;
}
location / {
    set $memc_cmd $arg_cmd;
    set $memc_key $arg_key;
    set $memc_value $arg_val;
    set $memc_flags $arg_flags;
    set $memc_exptime $arg_exptime;

    memc_cmds_allowed get set add incr delete flush_all;
    memc_pass memcached;
}
error_page 500 502 503 504 /50x.html;
location = /50x.html {
    root html;
}
}
}

```

## 2. 测试配置文件，并重启服务

```

# /usr/local/openresty/nginx/sbin/nginx -t -p /usr/local/openresty/nginx/
nginx: the configuration file /usr/local/openresty/nginx/conf/nginx.conf syntax
is ok
nginx: configuration file /usr/local/openresty/nginx/conf/nginx.conf test is
successful

```

```

/usr/local/openresty/nginx/sbin/nginx -p /usr/local/openresty/nginx/ -s reload

```

## 3. 测试结果

```

# curl 'localhost/?cmd=set&key=1&val=test'
STORED

# curl 'localhost/?cmd=get&key=1'
test

# curl 'localhost/?cmd=delete&key=1'
DELETED

# curl 'localhost/?cmd=add&key=2&val=100'
STORED

# curl 'localhost/?cmd=get&key=2'
100

# curl 'localhost/?cmd=incr&key=2&val=1'
101

# curl 'localhost/?cmd=incr&key=2&val=1'
102

```

```
# curl 'localhost/?cmd=decr&key=2&val=1'
<html>
<head><title>403 Forbidden</title></head>
<body bgcolor="white">
<center><h1>403 Forbidden</h1></center>
<hr><center>ngx_openresty/1.2.4.14</center>
</body>
</html>
```

```
# curl 'localhost/?cmd=flush_all'
OK
```

```
# curl 'localhost/?cmd=get&key=1'
<html>
<head><title>404 Not Found</title></head>
<body bgcolor="white">
<center><h1>404 Not Found</h1></center>
<hr><center>ngx_openresty/1.2.4.14</center>
</body>
</html>
```

```
# curl 'localhost/?cmd=get&key=2'
<html>
<head><title>404 Not Found</title></head>
<body bgcolor="white">
<center><h1>404 Not Found</h1></center>
<hr><center>ngx_openresty/1.2.4.14</center>
</body>
</html>
```

```
# curl 'localhost/memcached-status'
```

```
STAT pid 4914
STAT uptime 2774
STAT time 1360198988
STAT version 1.4.15
STAT libevent 2.0.21-stable
STAT pointer_size 64
STAT rusage_user 0.014997
STAT rusage_system 0.015997
STAT curr_connections 5
STAT total_connections 46
STAT connection_structures 6
STAT reserved_fds 20
STAT cmd_get 35
STAT cmd_set 8
STAT cmd_flush 5
STAT cmd_touch 0
STAT get_hits 11
STAT get_misses 24
STAT delete_misses 0
STAT delete_hits 1
STAT incr_misses 0
STAT incr_hits 4
STAT decr_misses 0
STAT decr_hits 0
```



```

STAT cas_misses 0
STAT cas_hits 0
STAT cas_badval 0
STAT touch_hits 0
STAT touch_misses 0
STAT auth_cmds 0
STAT auth_errors 0
STAT bytes_read 611
STAT bytes_written 1584
STAT limit_maxbytes 33554432
STAT accepting_conns 1
STAT listen_disabled_num 0
STAT threads 4
STAT conn_yields 0
STAT hash_power_level 16
STAT hash_bytes 524288
STAT hash_is_expanding 0
STAT bytes 0
STAT curr_items 0
STAT total_items 11
STAT expired_unfetched 1
STAT evicted_unfetched 0
STAT evictions 0
STAT reclaimed 1
END

```

## 13.2 lua-resty-memcached 访问方式

lua-resty-memcached 是 OpenResty 团队提供的 Memcached 访问库。lua-resty-memcached 适合在 Lua 代码里使用，可以灵活控制各个环节，并且节省内存，和内存池结合得更好。

### 13.2.1 概述

下面是一个 lua-resty-memcached 使用示例，演示了典型的使用方法。

```
lua_package_path "/path/to/lua-resty-memcached/lib/?.lua;;";
```

```

server {
    location /test {
        content_by_lua '
            local memcached = require "resty.memcached"
            local memc, err = memcached:new()
            if not memc then
                ngx.say("failed to instantiate memc: ", err)
                return
            end
            memc:set_timeout(1000) -- 1 sec
            -- or connect to a unix domain socket file listened
            -- by a memcached server:
            -- local ok, err = memc:connect("unix:/path/to/memc.sock")
            local ok, err = memc:connect("127.0.0.1", 11211)
            if not ok then

```

```

        ngx.say("failed to connect: ", err)
        return
    end
    local ok, err = memc:flush_all()
    if not ok then
        ngx.say("failed to flush all: ", err)
        return
    end
    local ok, err = memc:set("dog", 32)
    if not ok then
        ngx.say("failed to set dog: ", err)
        return
    end
    local res, flags, err = memc:get("dog")
    if err then
        ngx.say("failed to get dog: ", err)
        return
    end
    if not res then
        ngx.say("dog not found")
        return
    end
    ngx.say("dog: ", res)
    -- put it into the connection pool of size 100,
    -- with 10 seconds max idle timeout
    local ok, err = memc:set_keepalive(10000, 100)
    if not ok then
        ngx.say("cannot set keepalive: ", err)
        return
    end
    -- or just close the connection right away:
    -- local ok, err = memc:close()
    -- if not ok then
    --     ngx.say("failed to close: ", err)
    --     return
    -- end
}
}

```

### 13.2.2 API

下面的 API 中，key 参数在被传送给服务器之前将被自动按 URI 编码规则编码。

#### 1. new

语法：

```
memc, err = memcached:new(opts?)
```

说明：创建 Memcached 对象，失败则返回 nil 和描述字符串。opts 是可选参数，类型为表。支持下面的数据选项。

key\_transform：包含两个函数的数组，函数用于对 key 转码和解码。默认地，key 会使用 URI 组件转码和解码。

```
memcached:new{
    key_transform = { ngx.escape_uri, ngx.unescape_uri }
}
```

## 2. connect

语法:

```
ok, err = memc:connect(host, port)
ok, err = memc:connect("unix:/path/to/unix.sock")
```

说明: 连接到服务器。实际连接之前会先遍历连接池。

## 3. set

语法:

```
ok, err = memc:set(key, value, exptime, flags)
```

说明: 无条件插入一条记录。如果 key 已经存在, 将覆盖。

value 可以是一个 Lua 表, 容纳多个 Lua 字符串, 例如:

```
memc:set("dog", {"a ", {"kind of"}, " animal"})
```

等同于

```
memc:set("dog", "a kind of animal")
```

exptime 参数可选, 默认为 0。

flags 参数可选, 默认为 0。

## 4. set\_timeout

语法:

```
ok, err = memc:set_timeout(time)
```

说明: 设置超时值, 包括 connect 操作。成功则返回 1, 失败则返回 nil 和错误描述。

## 5. set\_keepalive

语法:

```
ok, err = memc:set_keepalive(max_idle_timeout, pool_size)
```

说明: 将对象立即放入连接池内, 对象进入 close 状态, 可用于原本使用 close 的地方。

参见其他模块的 set\_keepalive 部分。

## 6. get\_reused\_times

语法:

```
times, err = memc:get_reused_times()
```

说明: 返回当前连接重用次数, 用于判断是否是来自于连接池, 如果不是永远返回 0。

## 7. close

语法:

```
ok, err = memc:close()
```

说明: 关闭当前连接, 成功则返回 1, 失败则返回 nil 和错误描述。

## 8. add

语法:

```
ok, err = memc:add(key, value, exptime, flags)
```

说明: 插入一条记录, 只有 key 不存在才会成功。

参数同样可以是 Lua 表, 容纳多个 Lua 字符串, 例如:

```
memc:add("dog", {"a ", {"kind of"}, " animal"})
```

等同于

```
memc:add("dog", "a kind of animal")
```

exptime 参数可选, 默认为 0。

flags 参数可选, 默认为 0。

## 9. replace

语法:

```
ok, err = memc:replace(key, value, exptime, flags)
```

说明: 插入一条记录, 只有 key 存在才会成功。

参数同样可以是 Lua 表, 容纳多个 Lua 字符串, 例如:

```
memc:add("dog", {"a ", {"kind of"}, " animal"})
```

等同于

```
memc:add("dog", "a kind of animal")
```

exptime 参数可选, 默认为 0。

flags 参数可选, 默认为 0。

## 10. append

语法:

```
ok, err = memc:append(key, value, exptime, flags)
```

说明: 向 key 的 value 追加内容。

参数同样可以是 Lua 表, 容纳多个 Lua 字符串, 例如:

```
memc:add("dog", {"a ", {"kind of"}, " animal"})
```

等同于

```
memc:add("dog", "a kind of animal")
```

exptime 参数可选，默认为 0。

flags 参数可选，默认为 0。

### 11. prepend

语法：

```
ok, err = memc:prepend(key, value, exptime, flags)
```

说明：向 key 的 value 前面追加内容。

参数同样可以是 Lua 表，容纳多个 Lua 字符串，例如：

```
memc:add("dog", {"a ", {"kind of"}, " animal"})
```

等同于

```
memc:add("dog", "a kind of animal")
```

exptime 参数可选，默认为 0。

flags 参数可选，默认为 0。

### 12. get

语法：

```
value, flags, err = memc:get(key) syntax: results, err = memc:get(keys)
```

说明：从服务器获取 1 个或多个 key 的值。key 可以是单 key，也可以是存放在 Lua 表中的多 key。

单 key 的情况下，如果 key 找到了，返回 value 和 flags。发生错误时将返回 nil，flags 和 str 将返回描述性错误。

如果没有找到，则返回 3 个 nil。

多 key 的情况下，返回结果将保存在一个 Lua 表中返回。每一个 key 对应着一个表，存放 value 和 flags。如果 key 不存在，则没有对应的结果表。

任何错误发生时，将返回 nil 和错误描述信息。

### 13. gets

语法：

```
value, flags, cas_unique, err = memc:gets(key)
results, err = memc:gets(keys)
```

说明：跟 get 方法很像，只是返回 CAS 唯一值，本方法通常和 cas 方法一起使用。

### 14. cas

语法：

```
ok, err = memc:cas(key, value, cas_unique, exptime?, flags?)
```

说明：和 set 方法相似，只是做一个检查和 set 操作，意思是：如果从我上次获取这个值以来没有人修改过，那么存储这个数据。

cas\_unique 参数可以从 gets 方法获取。

### 15. touch

语法：

```
ok, err = memc:touch(key, exptime)
```

说明：更新一个已经存在 key 的过期时间。

### 16. flush\_all

语法：

```
ok, err = memc:flush_all(time?)
```

说明：清掉服务中的所有元素，不指定 time 则默认立即操作，指定 time 则在 time 时间后操作（秒）。

### 17. delete

语法：

```
ok, err = memc:delete(key)
```

说明：立即删除一个 key。key 必须已经存在于服务中。

### 18. incr

语法：

```
new_value, err = memc:incr(key, delta)
```

说明：将 key 的值按 delta 里的整数增长。成功则返回增长后的新值，失败则返回 nil 和错误描述。

### 19. decr

语法：

```
new_value, err = memc:decr(key, value)
```

说明：将 key 的 value 减少 value。成功则返回新文值，失败则返回 nil 和错误描述。

### 20. stats

语法：

```
lines, err = memc:stats(args?)
```

说明：返回服务器的统计信息。成功则返回一个 Lua 表，容纳所有的行。失败则返回 nil 的错误描述。args 参数未指定，返回服务器统计信息，args 可以是 sizes、slabs 或其他值。

## 21. version

版本:

```
version, err = memc:version(args?)
```

说明: 返回服务器版本号, 如 1.2.8。

## 22. quit

语法:

```
ok, err = memc:quit()
```

说明: 关闭当前连接。

## 23. verbosity

语法:

```
ok, err = memc:verbosity(level)
```

说明: 设置冗余级别, level 必须是数值型。

### 13.2.3 自动日志

如果自行处理了错误日志, 需要关闭自动日志功能:

```
lua_socket_log_errors off;
```

### 13.2.4 限制

使用 lua-resty-memcached 库时要注意以下两点限制:

- 本库不能在 `set_by_lua*`、`log_by_lua*`、`header_filter_by_lua*` 上下文中使用, 因为 `cosocket API` 处于无效状态。
- `resty.memcached` 对象不能存储在模块级变量中, 只能保存在局部变量或 `ctx.ctxtable` 中。

## 13.3 小结

在 Lua 中访问 Memcached, 有 `mem-nginx-module` 和 `lua-resty-memcached` 两种常用方法。`ngx_memc` 使用简单, 通过连接池的配置, 可以获得不错的性能。`lua-resty-memcached` 使用更灵活, 更节约内存。两种方式需要根据具体项目的模型和用途选择。

## PostgreSQL 操作

ngx\_postgres 用于和 PostgreSQL 数据库通信。应答包是 RDS 格式，可以和 rds-json-nginx 模块的 drizzle-nginx-module 一起工作。

默认在 OpenResty 下没有使能本模块，需要在编译 OpenResty 时使用 `--with-http_postgres_module` 选项打开。本模块需要系统中首先安装 libpq。

### 14.1 概述

下面是一个 ngx\_postgre 使用示例，这个示例向我们展示了 ngx\_postgre 的使用概貌。RDS 格式的 cats 表结果集：

```
http {
    upstream database {
        postgres_server 127.0.0.1 dbname=test user=test password=test;
    }

    server {
        location / {
            postgres_pass database;
            postgres_query "SELECT * FROM cats";
        }
    }
}
```

示例中定义了一个 location /，向 cats 表发起了一个查询。查询通过 postgres\_query 发起，请求通过 postgres\_pass 传递到 database 这个 upstream，而服务器的信息配置在 postgres\_server 这条配置里。



这是一个典型的 upstream 访问模式。

## 14.2 配置指令

ngx\_postgres 是一个通过配置指令实现的 upstream 模块。本节介绍所有配置指令。

### 1. postgres\_server

语法：

```
postgres_server ip[:port] dbname=dbname user=user password=pass
```

默认：none。

上下文：upstream。

说明：设置要使用的数据库参数。

### 2. postgres\_keepalive

语法：

```
postgres_keepalive off | max=count [mode=single|multi] [overflow=ignore|reject]
```

默认：max=10 mode=single overflow=ignore。

上下文：upstream。

说明：配置连接池参数如下。

- max：每个工作进程最多连接数。
- mode：后端匹配模式。
- overflow：ignore 表示忽略连接池满了的情况并且允许请求，但之后关闭连接，或直接使用 503 错误拒绝请求。

### 3. postgres\_pass

语法：

```
postgres_pass upstream
```

默认：none。

上下文：location、if location。

说明：设置数据连接使用的 upstream 块，可以包含变量。

### 4. postgres\_query

语法：

```
postgres_query [methods] query
```

默认：none。

上下文：http、server、location、if location。

说明：设置查询语句，可以包含变量。如果 `methods` 变量指定了，则查询只使用于 `methods` 方法，否则可以使用于所有的方法。

本指令可以在相同的上下文内使用多次。

### 5. postgres\_rewrite

语法：

```
postgres_rewrite [methods] condition [=]status_code
```

默认：none。

上下文：http、server、location、if location。

当条件成立时，重写应答状态码：

- `no_changes`：本查询没有受影响的数据行。
- `changes`：至少一行被查询影响到。
- `no_rows`：结果集没有返回一行数据。
- `rows`：结果集里至少返回一行。

`status_code` 使用 “=” 前缀时表示原始应答包体代替 `status_code` 表示的默认错误页发送给请求方。

`no_changes` 和 `changes` 只使用于 INSERT、UPDATE、DELETE、MOVE、FETCH 和 COPY 查询。

在相同的上下文中，本指令可以使用多次。

### 6. postgres\_output

语法：

```
postgres_output rds|text|value|binary_value|none
```

默认：rds。

上下文：http、server、location、if location。

说明：设置输出格式。

- `rds`：结果集使用 RDS 格式。
- `text`：使用文本格式（使用默认的 Content-Type），值使用新行分隔。
- `value`：使用文本格式返回单值（使用默认的 Content-Type）。
- `binary_value`：使用二进制格式（使用默认的 Content-Type）。
- `none`：不返回任何数据，只在使用了 `postgres_set` 设置其他模块处理结果集时使用默认的 Content-Type。

### 7. postgres\_set

语法：

```
postgres_set $variable row column [optional|required]
```

默认: none。

上下文: http、server、location。

说明: 从结果集中读取单值, 保存值在 \$variable。

当请求级别设置为 required 并且值超出范围、null 或 0 长度时, Nginx 返回 500 错误。

当需要的级别是 optional 时, 这种情况会被忽略。

行和列数从 0 开始。列号可以用列名代替。

本指令可以在相同的上下文中调用多次。

## 8. postgres\_escape

语法:

```
postgres_escape $escaped [[=]$unesescaped]
```

默认: none。

上下文: http、server、location。

说明: 解码 \$unesescaped 字符串, 结果保存在 \$escaped 变量中, 可以安全地在 SQL 查询中使用。

因为 Nginx 不能描述空和不存在字符串, 所有空字符中默认地转成 null 值。可以通过在 \$unesescaped 前面加 “=” 禁用。

## 9. postgres\_connect\_timeout

语法:

```
postgres_connect_timeout timeout
```

默认: 10s。

上下文: http、server、location。

说明: 设置连接到服务器的超时值。

## 10. postgres\_result\_timeout

语法:

```
postgres_result_timeout timeout
```

默认: 30s。

上下文: http、server、location。

说明: 设置接收超时值。

# 14.3 配置变量

配置指令支持下列变量, 可以在配置过程中使用。

- \$postgres\_columns: 结果集中的列数量。

- \$postgres\_rows: 结果集中的行数量。
- \$postgres\_affected: 受影响的行数, 由 INSERT、UPDATE、DELETE、MOVE、FETCH、COPY 查询产生。
- \$postgres\_query: SQL 查询语句。

## 14.4 示例

这里提供几个示例方便对 ngx\_postgres 模块学习。

示例 1: 从 sites 表返回从 \$http\_host 变量匹配的行。

```
http {
    upstream database {
        postgres_server 127.0.0.1 dbname=test user=test password=test;
    }

    server {
        location / {
            postgres_pass database;
            postgres_query SELECT * FROM sites WHERE host='$http_host';
        }
    }
}
```

这个示例定义了 location/ 以供其他请求使用, 这是一个子请求或 REST 接口, 使用 \$http\_host 这个变量作为查询参数, 读取信息。请求配置到 database 这个 upstream 上。

示例 2: 将请求转发给从数据中选择的后端点。

```
http {
    upstream database {
        postgres_server 127.0.0.1 dbname=test user=test password=test;
    }

    server {
        location / {
            eval_subrequest_in_memory off;

            eval $backend {
                postgres_pass database;
                postgres_query "SELECT * FROM backends LIMIT 1";
                postgres_output value 0 0;
            }

            proxy_pass $backend;
        }
    }
}
```

这个示例使用 eval 模块将请求使用 proxy\_pass 指令向上游请求适配, 本示例需要模块 nginx-eval-module (agntzh's fork) 支持。

示例 3: 使用本地数据库进行校验, 实现访问限制。

```
http {
    upstream database {
        postgres_server 127.0.0.1 dbname=test
                        user=test password=test;
    }

    server {
        location = /auth {
            internal;

            postgres_escape $user $remote_user;
            postgres_escape $pass $remote_passwd;

            postgres_pass database;
            postgres_query "SELECT login FROM users WHERE login=$user AND
                           pass=$pass";
            postgres_rewrite no_rows 403;
            postgres_output none;
        }

        location / {
            auth_request /auth;
            root /files;
        }
    }
}
```

这是一个比较典型的应用, 从数据库中获取校验信息, 实现访问控制, 这里可以加以扩展, 从 Redis 或 Memcached 中缓存访问控制信息, 则可以形成一个生产系统上可用的访问控制。

本示例需要下面模块支持:

- ngx\_http\_auth\_request\_module。
- ngx\_coolkit。

示例 4: 返回 JSON 的简单 RESTful Webservice。

```
http {
    upstream database {
        postgres_server 127.0.0.1 dbname=test
                        user=test password=test;
    }

    server {
        set $random 123;

        location = /numbers/ {
            postgres_pass database;
            rds_json on;

            postgres_query HEAD GET "SELECT * FROM numbers";
        }
    }
}
```

```

    postgres_query POST"INSERT INTO numbers VALUES('$random') RETURNING *";
    postgres_rewrite POST      changes 201;

    postgres_query DELETE      "DELETE FROM numbers";
    postgres_rewrite DELETE no_changes 204;
    postgres_rewrite DELETE changes 204;
}

location ~ /numbers/(?<num>\d+) {
    postgres_pass database;
    rds_json on;

    postgres_query HEAD GET "SELECT * FROM numbers WHERE number='$num'";
    postgres_rewrite HEAD GET no_rows 410;

    postgres_query PUT"UPDATE numbers SET number='$num' WHERE number
                        = '$num' RETURNING *";
    postgres_rewrite PUT no_changes 410;

    postgres_query DELETE "DELETE FROM numbers WHERE number='$num'";
    postgres_rewrite DELETE no_changes 410;
    postgres_rewrite DELETE changes 204;
}
}

```

这个示例使用了大部分的配置指令，演示了比较复杂的应用。通常这个模块还适合应用于简单的使用场景，可以直接实现 REST 接口。本示例需要 ngx\_rds\_json 模块支持。

示例 5：在 SQL 查询中使用 GET 参数。

```

location /quotes {
    set_unescape_uri $txt $arg_txt;
    postgres_escape $txt;
    postgres_pass database;
    postgres_query "SELECT * FROM quotes WHERE quote=$txt";
}

```

这个示例只是增加了 set\_unescape\_uri 指令和 postgres\_escape 指令，对输入处理做了 URI 解码。

本示例需要 ngx\_set\_misc 模块支持。

## 14.5 小结

ngx\_postgres 库集成在 OpenResty 包中，但默认没有启用，需要通过对应的编译参数启用。ngx\_postgres 库通过配置指令实现，使用比较简单。

## MongoDB 操作

Nginx 下使用 lua-resty-mongol 模块访问 MongoDB，但 lua-resty-mongol 模块并没有打包在 OpenResty 内，需要单独安装，即在 OpenResty 的 lualib 目录下复制进对应的脚本文件。lua-resty-mongol 库提供了很好的性能和灵活性，内存占用率低。

### 15.1 安装

首先从 git 上下载 lua-resty-mongol。如果机器上没有 git，则要先安装 git：

```
yum -y install git
```

然后执行下面的命令：

```
git clone https://github.com/bigplum/lua-resty-mongol.git
cd lua-resty-mongol
make PREFIX=/usr/local/openresty install
```

/usr/local/openresty 为已经安装的 resty 的安装路径，安装程序会在 /usr/local/openresty/lualib/resty 目录下面建立 mongol 目录，并添加 mongol 的 Lua 脚本。

### 15.2 配置

在 nginx.conf 中添加库目录：

```
lua_package_path '/usr/local/openresty/lualib/?.init.lua;;';
```

或者在 Lua 文件使用之前调用：

```
local p = "/usr/local/openresty/lualib/"
local m_package_path = package.path
package.path = string.format("%s?.lua;%s?/init.lua;%s", p, p, m_package_path)
```

调用 require 操作会得到一个到 mongod 的连接对象函数，后面的操作都使用这个对象。

```
mongol = require "resty.mongol"
conn = mongol:new() -- return a connection object
```

## 15.3 操作函数

lua-resty-mongol 库函数共分为 3 类：①连接对象，用于管理和服务器的连接；②数据库对象方法，用于获取数据行；③列对象，用于对数据行进行分析和处理。三者的关系如下：连接对象返回数据库对象，数据库对象返回列，列返回需要访问的数据。

### 15.3.1 连接对象方法

连接对象操作包含以下方法：

1. `ok,err = conn:connect(host, port)`

功能：连接到服务器，默认值 host 是 localhost，端口是 27017。

2. `ok,err = conn:set_timeout(msec)`

功能：设置套接字超时值，影响 connect、read、writing，单位是毫秒。

成功则返回 1，失败则返回 nil 和错误描述字符串。

3. `ok,err = conn:set_keepalive(msec, pool_size)`

功能：将连接放入连接池，成功则返回 1，失败则返回 nil 和错误描述字符串。

4. `times,err = conn:get_reused_times()`

功能：返回连接重用次数。

5. `ok,err = conn:close()`

功能：关闭连接。

6. `bool, hosts = conn:ismaster()`

功能：返回一个布尔值标示是否这是一个主服务器，并且返回一个其他主机的表 hosts，失败则返回 nil 和 err。

7. `newconn = conn:getprimary ([already_checked])`

功能：返回一个连接到主服务器的连接，或者返回 nil 和 errmsg。返回的连接对象可能就是它自己。

8. `databases = conn:databases()`

功能：返回一个服务器上数据库的表。



```
databases.name: string
databases.empty: boolean
databases.sizeOnDisk: number
```

#### 9. conn:shutdown()

功能：关闭服务器，没有返回。

#### 10. db = conn:new\_db\_handle(database\_name)

功能：返回一个数据库对象，或者 nil。

### 15.3.2 数据库对象方法

数据库对象操作包含以下方法：

#### 1. db:list()

功能：列出数据库所有表。

#### 2. db:dropDatabase()

功能：删除数据库。

#### 3. db:add\_user(username, password)

功能：添加新用户。

#### 4. ok, err = db:auth(username, password)

功能：校验用户，成功则返回 1，失败则返回 nil 和错误描述。

#### 5. col = db:get\_col(collection\_name)

功能：返回一个列对象，以便在上面执行更多操作。

#### 6. gridfs = db:get\_gridfs(fs)

功能：从 MongoDB 获取一个 gridfs 对象。

### 15.3.3 列对象方法

列对象操作包含以下方法：

#### 1. n = col:count(query)

功能：返回列表内 query 条件结果数。

#### 2. ok, err = col:drop()

功能：删除当前列。

#### 3. n, err = col:update(selector, update, upsert, multiupdate, safe)

功能：返回被更新的列数或失败时返回 nil。

- upsert 设置为 1 时，如果没有找到匹配的文档，照样把支持的对象插入数据库。默认是 0。
- multiupdate 设置为 1 时，数据库将所有匹配的对象进行更新，否则只更新第一个匹

配的记录。默认是 0。

- safe 可以是布尔值或整型值。默认是 0，如果为 1，则程序将向服务器发送一个 getlasterror 命令查询结果。如果是 false，则返回值 n 将总是 -1。

#### 4. n, err = col:insert(docs, continue\_on\_error, safe)

功能：如果设置了 continue\_on\_error 值，当执行一个块插入操作时，出现错误也不会停止执行（如重复的 ID）。

safe 的含义同第 3 条命令的 safe 参数。

#### 5. n, err = col:delete(selector, singleRemove, safe)

功能：返回被删除的行数，失败则返回 nil 和错误描述。

- singleRemove 设置为 1 时，数据库只会删除第一个匹配的记录，否则所有的记录会被删除。默认为 0。
- safe 的含义同第 3 条命令的 safe 参数。

#### 6. r = col:find\_one(query, returnfields)

功能：返回一个单元数据，否则返回 nil。

returnfields 是需要返回的域，如 {n=0} 或 {n=1}。

#### 7. cursor = col:find(query, returnfields, num\_each\_query)

功能：为查询语句返回一个游标对象。

- returnfields：需要返回的域，如 {n=0} 或 {n=1}。
- num\_each\_query：游标每次查询的最大返回数，必须大于 1，0 是无限限制，默认是 100。

#### 8. col:getmore(cursorID, [numberToReturn], [offset\_i])

- cursorID：一个 8 字节字符串。
- numberToReturn：要返回多少结果，默认是 -1。
- offset\_i：开始的数量，默认为 1。

#### 9. col:kill\_cursors(cursorIDs)

功能：释放当前游标。

## 15.4 示例

示例 1：实现简单的键值读取，这是最常用的使用情况，同样可以实现将我们的键值数据保存到数据库上。

```
local mongo = require "resty.mongol"
conn = mongo:new()
conn:set_timeout(1000)
ok, err = conn:connect()
```

```

if not ok then
    ngx.say("connect failed: "..err)
end

local db = conn:new_db_handle ( "test" )
col = db:get_col("test")

r = col:find_one({name="dog"})
ngx.say(r["name"])

```

或进行进一步解析：

```

local mongo = require "resty.mongol"
local conn = mongo:new()
conn:set_timeout(1000)
local ok, err = conn:connect("127.0.0.1",27017)

```

```

if not ok then
    ngx.say("connect failed: «..err)
end

```

```

local db=conn:new_db_handle("test")
local col = db:get_col("test")
local r = col:find_one({name="dog"},{_id=0})

```

```

for k,v in pairs(r) do
    ngx.say(k..": «..v)
end

```

示例 2：将 MongoDB 返回的 BSON 格式结果集，用 CJSON 库转换为 JSON 返回给 Web 使用 JavaScript 处理。

```

local mongo = require "resty.mongol"
local json = require("cjson")
local conn = mongo:new()
conn:set_timeout(1000)

```

```

local ok, err = conn:connect("127.0.0.1",27017)

```

```

if not ok then
    ngx.say("connect failed: «..err)
end

```

```

local db=conn:new_db_handle("meedo-service")
local col = db:get_col("channels")
local r = col:find_one({_id=1})

```

```

value = json.encode(r)
ngx.say(value)

```

## 15.5 小结

使用 lua-resty-mongol 库访问 MongoDB，但 lua-resty-mongol 并没有打包在 OpenResty 内，需要单独安装。lua-resty-mongol 与 OpenResty 团队开发的其他库拥有相同的访问网络，性能好，内存占用率非常小。

## bit 库的使用

Lua 提供了 bit 库，可以对变量数据进行位运算，在某些应有场景，我们需要在 Lua 中对数据进行位移，或进行“与、或、非”及进制转换等操作。

例如，用一个 32 位的整数表示 RGB 颜色。32 位整数，被分为 4 个部分，每个部分 8 位，8 位可表示的十进制数的范围是 0 ~ 255。

## 16.1 示例

下面演示爱拉托逊斯筛法，计算一定范围内质数的数量。使用一个 Lua 表容纳位 vector。每一个数组索引有 32 位的 vector。位操作用于访问和操作这些容器和元素。注意，不需要标记移位数量。

```
local bit = require("bit")
local band, bxor = bit.band, bit.bxor
local rshift, rol = bit.rshift, bit.rol

local m = tonumber(arg and arg[1]) or 100000
if m < 2 then m = 2 end
local count = 0
local p = {}

for i=0,(m+31)/32 do p[i] = -1 end

for i=2,m do
  if band(rshift(p[rshift(i, 5)], i), 1) ~= 0 then
    count = count + 1
    for j=i+i,m,i do
      local jx = rshift(j, 5)
```

```

p[jx] = band(p[jx], rol(-2, j))
end
end
end
io.write(string.format("Found %d primes up to %d\n", count, m))

```

LuaBitOp 操作非常快。这个程序在 3GHz CPU 上的运行时间少于 90 毫秒，但是事实上执行了超过 100 万次位操作。这是在标准 Lua 安装上得到的数据，如果使用 LuaJIT，速度会更快。

## 16.2 安装

因为 bit 库需要使用到 lua-devel 库，所以需要在 CentOS 上首先安装 lua-devel 库。

安装 lua-devel 库：

```
yum install lua-devel.*
```

下载 BitOp 源码包：

```
wgethttp://bitop.luajit.org/download/LuaBitOp-1.0.1.tar.gz
```

解压缩：

```
tar -xzf LuaBitOp-1.0.2.tar.gz
cd LuaBitOp-1.0.2
```

编译：

```
make (编译出来的库在目录下为 bit.so)
make install
```

bit.so 将安装在 lua C 库目录下，如 /usr/lib64/lua/5.1/。

## 16.3 函数

BitOp 提供了大部分的位操作函数。

### 1. 载入 BitOp 模块

推荐使用 require 命令载入 BitOp 模块：

```
local bit = require("bit")
```

这个方法限制了访问范围在当前文件，同时提供快速的方式访问 bit.\* 函数。不在全局变量使用模块对象是好的编程实践。require 函数保证模块在任何情况下只会载入一次。

### 2. 定义快捷方式

通用的做法是将常用的模块函数缓冲到 local 变量内。这种方法将加速这些函数解析。

```
local bnot = bit.bnot
```

```
local band, bor, bxor = bit.band, bit.bor, bit.bxor
local lshift, rshift, rol = bit.lshift, bit.rshift, bit.rol
-- etc...
```

-- 快捷方式示例

```
local function tr_i(a, b, c, d, x, s)
returnrol(bxor(c, bor(b, bnot(d))) + a + x, s) + b
end
```

and、or 和 not 是 Lua 保留关键字，它们不能用于变量名和语义域名。所以位函数命名为 band、bor 和 bnot (bxor 异或)。

### 3. Bit 操作

printfx 函数将参数作为一个无符号 32 位十六进制数输出，后面的函数示例中将使用本函数。

```
function printfx(x)
print("0x"..bit.tohex(x))
end
```

#### (1) y = bit.tobit(x)

功能：将一个数值规范化，做好位操作准备并返回。

并不是在所有的位操作之前都需要对参数进行规范化操作，具体需要查看每个操作的说明。

```
print(0xffffffff)      --> 4294967295 (*)
print(bit.tobit(0xffffffff)) --> -1
printfx(bit.tobit(0xffffffff)) --> 0xffffffff
print(bit.tobit(0xffffffff + 1)) --> 0
print(bit.tobit(2^40 + 1234)) --> 1234
```

#### (2) y = bit.tohex(x [,n])

功能：将 x 转换为十六进制字符串。

n 是可选参数，表示生成的十六进制字节数。正数 1 ~ 8 的取值会生成小写的十六进制串。负数取值将会生成大写十六进制串。结果数值只使用重要的 4|n| 位。默认生成 8 位小写十六进制数值。

```
print(bit.tohex(1))      --> 00000001
print(bit.tohex(-1))     --> ffffffff
print(bit.tohex(0xffffffff)) --> ffffffff
print(bit.tohex(-1, -8)) --> FFFFFFFF
print(bit.tohex(0x21, 4)) --> 0021
print(bit.tohex(0x87654321, 4)) --> 4321
```

#### (3) y = bit.bnot(x)

功能：按位取反。

```
print(bit.bnot(0))      --> -1
printfx(bit.bnot(0))    --> 0xffffffff
print(bit.bnot(-1))     --> 0
```

```
print(bit.bnot(0xffffffff))    --> 0
printx(bit.bnot(0x12345678))   --> 0xedcba987
```

(4)  $y = \text{bit.bor}(x1 [,x2\dots])$ ,  $y = \text{bit.band}(x1 [,x2\dots])$ ,  $y = \text{bit.bxor}(x1 [,x2\dots])$

功能：分别为或、与、异或操作。

```
print(bit.bor(1, 2, 4, 8))      --> 15
printx(bit.band(0x12345678, 0xff)) --> 0x00000078
printx(bit.bxor(0xa5a5f0f0, 0xaa55ff00)) --> 0x0ff00ff0
```

(5)  $y = \text{bit.lshift}(x, n)$ ,  $y = \text{bit.rshift}(x, n)$ ,  $y = \text{bit.arshift}(x, n)$

功能：分别为逻辑左移、逻辑右移、算术右移， $n$  表示移动位数， $n$  取值为 0 ~ 31。

逻辑移动将  $x$  作为无符号数操作，以 0 位填充。算术右移将移动的位作为有符号位复制并填充。

```
print(bit.lshift(1, 0))         --> 1
print(bit.lshift(1, 8))        --> 256
print(bit.lshift(1, 40))       --> 256
print(bit.rshift(256, 8))      --> 1
print(bit.rshift(-256, 8))     --> 16777215
print(bit.arshift(256, 8))     --> 1
print(bit.arshift(-256, 8))    --> -1
printx(bit.lshift(0x87654321, 12)) --> 0x54321000
printx(bit.rshift(0x87654321, 12)) --> 0x00087654
printx(bit.arshift(0x87654321, 12)) --> 0xffff87654
```

(6)  $y = \text{bit.rol}(x, n)$ ,  $y = \text{bit.ror}(x, n)$

功能：分别为左旋转、右旋转。 $n$  为旋转位数，范围为 0 ~ 31。

```
printx(bit.rol(0x12345678, 12)) --> 0x45678123
printx(bit.ror(0x12345678, 12)) --> 0x67812345
```

(7)  $y = \text{bit.bswap}(x)$

功能：交换  $x$  的字节，可以用来转换小端序 32 位数值为大端序 32 位数值。

```
printx(bit.bswap(0x12345678)) --> 0x78563412
printx(bit.bswap(0x78563412)) --> 0x12345678
```

## 16.4 说明

使用 BitOp 库时需要注意以下事项。

### 1. 有符号结果

避免使用 `string.format` 时用 “%x” 和 “%u” 进行格式化，因为在超过 8 个十六进制数字时这些格式会出错。

因为这些是有符号结果，所以也可能需要避免默认的数值向字符串转换，这个约束也适用于其他的接受字符串参数的标准库，如 `print()` 或 `io.write()`。

## 2. 条件

如果从 C/C++ 代码翻译成 Lua，要小心位操作为判断条件的情况。在 C/C++ 中非 0 值被当成 true 对待，例如：

```
if (x & 3) ...
```

不能这样转换为 Lua 代码：

```
if band(x, 3) then ... -- wrong!
```

在 Lua 中所有的对象除了 nil 都被认为是 true，包括所有的数值。所以这样处理是正确的：

```
if band(x, 3) ~= 0 then ... -- correct!
```

## 3. 十六进制比较

对位操作的结果（有符号数）和十六进制数（无符号）进行比较需要一些额外的考虑，下面的条件表达式可能不能正确工作，这依赖于平台：

```
bit.bor(x, 1) == 0xffffffff
```

在默认的数值类型下，这个条件永远不会为 true。

简单的方案如下：

- 不要在表达式中使用超过 0x7ffffff 的十六进制串：

```
bit.bor(x, 1) == -1
```

- 在比较之前使用 bit.tobit() 进行转换：

```
bit.bor(x, 1) == bit.tobit(0xffffffff)
```

- 为 bit.bxor() 生成一个工作区：

```
bit.bxor(bit.bor(x, 1), 0xffffffff) == 0
```

- 使用一个具体的工作区：

```
bit.rshift(x, 1) == 0x7ffffff
```

## 16.5 小结

因为 Lua 在语言一级并不支持位操作，不能像 C/C++ 等语言直接使用位操作，需要使用 bit 库进行位操作。本章详细介绍了 BitOp 库，使用 BitOp 库可以执行与 C/C++ 相同的位操作。位操作性能通常都要好一些。



## lfs 库的使用

lfs-file system operations (LuaFileSystem) 用于补充标准 Lua 发布版本的文件操作函数。LuaFileSystem 提供了一个访问底层目录文件属性的轻便方法。Lua 系统库里的文件函数只能打开文件以及对文件进行读写，lfs 库提供了对文件和目录操作的方法。

## 17.1 目录迭代示例

示例：对一个目录进行迭代并且递归显示每一个文件的属性。

```
require "lfs"

function attrdir (path)
    for file in lfs.dir(path) do
        if file ~= "." and file ~= ".." then
            local f = path..'/'..file
            print ("\t\t"..f)
            local attr = lfs.attributes (f)
            assert (type(attr) == "table")
            if attr.mode == "directory" then
                attrdir (f)
            else
                for name, value in pairs(attr) do
                    print (name, value)
                end
            end
        end
    end
end

attrdir (".")
```

## 17.2 安装

OpenResty 中并未打包 LuaFileSystem，所以使用 lfs 时，需要首先下载和安装。

从 <http://luaforge.net/projects/luafilesystem/files> 下载 lfs 源码，执行 configure、make、make install。或者将编译出来的 lfs.so 复制到 lua 库目录中，如 /usr/local/nginx/lib。

```
git clone https://github.com/keplerproject/luafilesystem
cd luafilesystem
make
make install
```

lfs.so 会被编译连接并复制到 lua 目录下，如 /usr/local/lib/lua/5.1。

## 17.3 LuaFileSystem 函数

LuaFileSystem 提供的文件和路径操作函数与 C 标准库中的对应函数非常相似，理解和使用起来无难度。

### 1. lfs.attributes

语法：

```
lfs.attributes (filepath [, aname])
```

说明：返回 filepath 对应文件属性的 table（或发生任何错误时返回 nil 和对应的错误描述）。如果第二个可选参数给定，则只返回给定的属性（等同于 fs.attributes(filepath).aname，但是不会创建属性结果表并且只会从操作系统返回一个属性）。属性在下面描述，属性模式是字符串，其他是数值，属性中使用的时间跟 os.time() 的时间格式相同。

- dev：在 UNIX 系统上，返回 inode 驻留的设备；在 Windows 系统上，描述包含文件的磁盘号。
- ino：在 UNIX 系统上，返回 inode 号；在 Windows 系统上，此模式无意义。
- mode：以字符串描述的保护模式（值可以是文件、路径、连接、套接字、命名管道、字符设备、块设备或其他）。
- nlink：文件硬连接号。
- uid：所有者的 user-id（UNIX 下有效，Windows 下总为 0）。
- gid：所有者的 group-id of owner（UNIX 下有效，Windows 下总为 0）。
- rdev：在 UNIX 系统上，描述设备类型；在 Windows 系统上与 dev 相同。
- access：上次访问的时间。
- modification：上次编辑的时间。
- change：上次状态改变的时间。
- size：字节单位的文件尺寸。

- blocks: 文件申请的块 (UNIX 下有效)。
- blksize: 最优文件 I/O 块尺寸 (UNIX 下有效)。

## 2. lfs.chdir

语法:

```
lfs.chdir (path)
```

说明: 改变当前工作路径到 path。成功则返回 true, 失败则返回 nil 和错误字符串。

## 3. lfs.currentdir

语法:

```
lfs.currentdir ()
```

说明: 返回当前工作路径的字符串, 失败则返回 nil 和错误字符串。

## 4. lfs.dir

语法:

```
lfs.dir (path)
```

说明: 迭代显示 path 目录的条目。每次调用, 都会返回一个条目字符串; 当没有更多条目时, 返回 nil。如果目录是空的, 引发一个错误。

## 5. lfs.lock

语法:

```
lfs.lock (filehandle, mode[, start[, length]])
```

说明: 锁定一个文件或文件的一部分。这个函数工作在打开的文件上; 文件句柄必须作为第一个参数。mode 是字符串模式, 可以是 r (读 / 共享锁) 或 w (写 / 独占锁)。可选参数 start 和 length 可以用来指定开始点和长度, 都为数值类型。

操作成功则返回 true, 任何错误发生则返回 nil 和错误字符串。

## 6. lfs.mkdir

语法:

```
lfs.mkdir (dirname)
```

说明: 创建一个新路径。操作成功则返回 True, 失败则返回 nil 和错误字符串。

## 7. lfs.rmdir

语法:

```
lfs.rmdir (dirname)
```

说明: 移除一个存在的路径。成功则返回 true, 失败则返回 nil 和错误字符串。

## 8. lfs.touch

语法:

```
lfs.touch (filepath [, atime [, mtime]])
```

说明: 设置文件的访问和编辑时间。这个函数是一个 `utime` 的绑定函数。第一个参数 `filepath` 是文件名, 第二个参数 `atime` 是访问时间, 第三个参数 `mtime` 是编辑时间。两个时间均以秒为单位 (通过 `os.date` 产生的时间)。如果忽略编辑时间, 则使用访问时间; 如果两个时间都忽略了, 则使用当前时间。

成功则返回 `true`, 失败则返回 `nil` 和错误字符串。

## 9. lfs.unlock

语法:

```
lfs.unlock (filehandle[, start[, length]])
```

说明: 解锁一个文件或文件的一部分。函数工作在一个打开的文件上, 文件句柄必须是第一个参数。可选参数 `start` 和 `length` 可以用来指定开始点和长度, 必须是数值型。

成功则返回 `true`, 失败则返回 `nil` 和错误字符串。

## 17.4 小结

Lua 的系统 OS 库提供了文件读写操作, `LuaFileSystem` 补充了系统库对文件属性和文件 / 路径操作。`lfs` 比较简单, 但对于我们需要对目录和文件进行操作的情况下比较有用, 例如, 可以用于创建和管理自己的文件缓存系统。

## resty.http 库的使用

resty.http 用于访问外部 HTTP 资源、location 等，如访问非本地 location、外部 Web 服务、RESTful、Web Service 等。resty.http 是一个轻量级 HTTP 库。

### 18.1 安装

下载源码：

```
git clone https://github.com/pint-sized/lua-resty-http
```

安装：

本库使用 Lua 编写，直接将下载目录中的 lib/resty 目录和上一层的 util 目录复制到 Lua 目录中，如 /usr/local/nginx/lib/，或 /usr/local/openresty/lualib/。openresty/lualib 下已经有 resty 目录了，复制文件即可。

resty.http 库支持：

- HTTP-1.0 /1.1。
- SSL。
- 应答包体流式接口，可预期式内存使用。
- 其他单请求无须手动连接可选的简单接口。
- 块式或非块式传输编码。
- 连接池。
- 流水线。
- “拖车”数据。

## 18.2 概述

示例: resty.http 库的使用方法和概况。

```
lua_package_path "/path/to/lua-resty-http/lib/?.lua;;";
server {
    location /simpleinterface {
        resolver 8.8.8.8; # 使用谷歌开放 DNS 服务器
        content_by_lua '

            -- 使用 URI 接口处理单请求
            local http = require "resty.http"
            local httpc = http.new()
            local res, err = httpc:request_uri("http://example.com/helloworld", {
                method = "POST",
                body = "a=1&b=2",
                headers = {
                    ["Content-Type"] = "application/x-www-form-urlencoded",
                }
            })

            if not res then
                ngx.say("failed to request: ", err)
                return
            end

            -- 这是简单的表单, 没有手动连接的步骤, 所以包体一次性读取
            -- 包括任意的 "拖车" 数据, 并且连接放到连接池里

            ngx.status = res.status

            for k,v in pairs(res.headers) do
                --
            end

            ngx.say(res.body)
        '
    }
}

location /genericinterface {
    content_by_lua '

        local http = require "resty.http"
        local httpc = http.new()

        -- 普通表单可以有更多控制, 需要手工连接
        httpc:set_timeout(500)
        httpc:connect("127.0.0.1", 80)

        -- 请求使用路径, 而不是完整的 URI
        local res, err = httpc:request{
            path = "/helloworld",
            headers = {
                ["Host"] = "example.com",
```

```

    },
  },
  if not res then
    ngx.say("failed to request: ", err)
    return
  end
  -- 可以使用 body_reader 迭代子了, 使用期望的块尺寸流化包体
  local reader = res.body_reader

  repeat
    local chunk, err = reader(8192)
    if err then
      ngx.log(ngx.ERR, err)
      break
    end

    if chunk then
      -- process
    end
  until not chunk

  local ok, err = httpc:set_keepalive()
  if not ok then
    ngx.say("failed to set keepalive: ", err)
    return
  end
end
';
}
}

```

## 18.3 函数

resty.http 提供了数量相对比较多的函数, 但接口都比较简单。

### 18.3.1 连接类

#### 1. new

语法:

```
httpc = http.new()
```

说明: 创建 HTTP 对象。失败则返回 nil 和错误描述。

#### 2. connect

语法:

```

ok, err = httpc:connect(host, port, options_table?)
ok, err = httpc:connect("unix:/path/to/unix.sock", options_table?)

```

说明：连接到 Web 服务器。

在实际解析主机名并连接到远程端点前，本方法总是先检查连接池寻找本方法之前创建的空闲连接。

`options_table`：可选的 Lua 表，用于指定连接设置。

`pool`：为使用的连接池指定一个自定义名字。如果省略本设置，连接池使用 `<host>:<port>` 或 `<unix-socket-path>` 模板生成。

### 3. set\_timeout

语法：

```
httpc:set_timeout(time)
```

说明：设置子操作（包括 `connect` 操作）的超时值，单位为毫秒。

### 4. ssl\_handshake

语法：

```
session, err = httpc:ssl_handshake(session, host, verify)
```

说明：在 TCP 连接上执行 SSL 握手操作，在 v0.9.11 版本以上才有效。

细节可参考第 28 章 `ngx.socket.tcp` 一节。

### 5. set\_keepalive

语法：

```
ok, err = httpc:set_keepalive(max_idle_timeout, pool_size)
```

说明：将当前连接放入 `ngx_lua` 的 `cosocket` 连接池中。

可以指定连接在连接池最大的空闲超时值（毫秒）和每一个 Nginx 工作进程中连接池尺寸。

本方法可以替换 `close` 方法，本方法可以立即将连接置入关闭状态。任意后续的子操作，如 `connect()` 都返回 `closed` 错误。

注意，使用本方法代替 `close` 是相对安全的，它会依赖请求的类型有条件地关闭。通常地，无连接的 1.0 请求——关闭，有连接的 1.1 请求——关闭。

调用成功则返回 1，发生任何错误则返回 `nil` 和错误描述。如果是上面的有条件关闭，则返回 2 和错误描述，并且连接会被关闭。

### 6. get\_reused\_times

语法：

```
times, err = httpc:get_reused_times()
```

说明：返回当前连接的重用次数，如果有错误，则返回 `nil` 和错误描述。

如果当前连接不是来自于连接池，将返回 0。如果连接来自于连接池，则返回非 0 值。所以，本方法可以用来判断连接是否来自于连接池。



## 7. close

语法:

```
ok, err = http:close()
```

说明: 关闭当前连接并返回状态。成功则返回 1, 失败则返回 nil 和错误描述。

## 8. request

语法:

```
res, err = httpc:request(params)
```

说明: 普通请求接口, 发起 HTTP 请求, 返回一个 res 表或当错误发生时返回 nil 和错误描述。

params 参数表接受下面的域:

- version: HTTP 版本, 1.0 或 1.1。
- method: HTTP 方法字符串。
- path: 路径字符串。
- query: 查询串。
- headers: 表, 存放请求头。
- body: 请求包体, 字符串, 或一个迭代函数 (参见 `get_client_body_reader`)。
- ssl\_verify SSL: 证书匹配的主机。

当请求成功, res 包含下面的域:

- status: 状态码。
- reason reason: 阶段。
- headers: 表, 存储 HTTP 头, 多个头域拥有相同的头域名称, 值要放在一个表中, 使头表中拥有一个头域。
- has\_body: 布尔值, 指示包体可以读取。
- body\_reader: 一个迭代子函数, 用于包体流式读取。
- read\_body: 一个方法, 将包体读入字符串。
- readtrailers: 一个方法, 在包体读取之后合并头域后面的拖车数据。

## 9. request\_uri

语法:

```
res, err = httpc:request_uri(uri, params)
```

说明: 简单接口 params 表支持的参数和普通接口一样, 并且将覆盖 URI 中找到的组件。

简单模式用于处理一个简单的 HTTP 请求, 没有太复杂的过程。本模式下, 不能采用流式应答包体。如果请求成功, res 将包含下面域:

- status: 状态码。

- headers: 头域表。
- body: 字符串式应答包体。

## 10. request\_pipeline

语法:

```
responses, err = httpc:request_pipeline(params)
```

说明: 本方法在前面的请求方法之上工作, params 代替参数表。每个请求按顺序发送, responses 是应答表, 例如:

```
local responses = httpc:request_pipeline{
  {
    path = "/b",
  },
  {
    path = "/c",
  },
  {
    path = "/d",
  }
}
for i,r in ipairs(responses) do
  if r.status then
    ngx.say(r.status)
    ngx.say(r:read_body())
  end
end
```

因为流水线的特性, 直到尝试读取应答域 (status/headers 等) 才实际读取应答, 而且在读取下一个应答前必须读取前一个应答的所有包体。

## 18.3.2 应答类

连接类函数返回 res 是应答类, 存储应答数据。

### 1. res.body\_reader

说明: body\_reader 可以使用自定义块尺寸流式化应答。例如:

```
local reader = res.body_reader
repeat
  local chunk, err = reader(8192)
  if err then
    ngx.log(ngx.ERR, err)
    break
  end
  if chunk then
    -- process
  end
until not chunk
```

如果未使用参数调用 reader, 具体行为依赖于连接的类型。如果应答以块式编码, 迭代

器将返回到达的块；否则，简单地返回整个包体。

## 2. res:read\_body

语法：

```
body, err = res:read_body()
```

说明：将整个包体读进一个本地字符串。

## 3. res:read\_trailers

语法：

```
res:read_trailers()
```

说明：合并头域表后的拖车数据，必须在包体读取之后读取。

### 18.3.3 代理类

有两个方便的方法可将当前请求代理到上游连接，安全地将下游数据发送到客户端，作为一个反向代理。

示例：

```
local http = require "resty.http" local httpc = http.new()
httpc:set_timeout(500) local ok, err = httpc:connect(HOST, PORT)
if not ok then
    ngx.log(ngx.ERR, err)
    return
end
httpc:set_timeout(2000)
httpc:proxy_response(httpc:proxy_request())
httpc:set_keepalive()
```

本示例共提供了两个函数实现代理操作。

#### 1. proxy\_request

语法：

```
local res, err = httpc:proxy_request(request_body_chunk_size?)
```

说明：使用当前请求参数执行一个请求，代理到连接上游。请求包体用流式读取。

#### 2. proxy\_response

语法：

```
httpc:proxy_response(res, chunksize?)
```

说明：使用 res 设置当前应答。确保头域发送到下游，并且依赖 chunksize 读取应答。

### 18.3.4 工具类

resty.http 库提供了两个工具类函数，实现包头读取和 URI 解析。

### 1. parse\_uri

语法:

```
local scheme, host, port, path, query? = unpack(httpc:parse_uri(uri, query_in_path?))
```

说明: 这是一个常用函数, 允许当输入 URI 时一对多使用普通接口。

query\_in\_path 参数指定是否查询串包含在返回值的路径中。默认为 true, 以保证后台兼容性。当设置为 false 时, path 只包含路径, query 将包含 URI 参数, 不包括?。

### 2. get\_client\_body\_reader

语法:

```
reader, err = httpc:get_client_body_reader(chunksize?, sock?)
```

说明: 返回迭代器函数, 用于流式读取下游客户端请求包体。可以指定可选的 chunksize (默认为 65536), sock 是一个和客户端请求建立的套接字。

例如:

```
local req_reader = httpc:get_client_body_reader()
repeat
    local chunk, err = req_reader(8192)
    if err then
        ngx.log(ngx.ERR, err)
        break
    end
    if chunk then
        -- process
    end
until not chunk
```

这个迭代器可以用来作为请求参数中包体域的值, 允许将一个请求包体流式化到代理的请求中, 例如:

```
local client_body_reader, err = httpc:get_client_body_reader()
local res, err = httpc:request{
    path = "/helloworld",
    body = client_body_reader,
}
```

## 18.4 小结

resty.http 是一个轻量级的 HTTP 库, 使用比较方便, 可以用于访问外部 RESTful 服务或 Web Service。

本章介绍了 resty.http 库的安装方法及库提供的所有方法, 并给出了使用示例。读者可以将示例修改为自己的使用代码。

## lcurl 库的使用

lcurl 库封装了 libcurl 的函数，为 Lua 提供了使用 curl 的接口。

libcurl 的主要功能是用不同的协议连接，与不同的服务器沟通，相当于封装了的 sock。libcurl 当前支持 http、https、ftp、gopher、telnet、dict、file、ldap 协议。libcurl 同样支持 HTTPS 证书授权，支持 HTTP POST、HTTP PUT、FTP 上传、HTTP 基本表单上传、代理、cookies 和用户认证。

lcurl 功能强大，可以为 Lua 提供方便而强大的网络访问能力。

### 19.1 安装

lcurl 库需要单独安装，才能使用。安装过程中因为依赖于 libcurl 库，所以需要分两步，先安装 libcurl 库，然后安装 lcurl 库。

#### 19.1.1 安装 libcurl

libcurl 库是源码编译，过程中需要用到 curl 库头文件，所以需要首先下载并安装 curl 库。

##### 1. 下载

从下载地址 <https://curl.haxx.se/download.html> 下载 curl-7.52.1.tar.gz 文件。

##### 2. 安装

```
tar -xzf curl-7.52.1.tar.gz
cd curl-7.52.1
make
make install
```

## 19.1.2 安装 lcurl

### (1) 下载

```
git clone https://github.com/moteus/lua-lcurl
```

### (2) 编译

```
cd lua-lcurl
make
```

### (3) 安装

将编译出来的 lcurl.so, 手动复制到 lua 目录下, 如 /usr/local/lib/lua/5.1; 将 src/lua 目录下的内容复制到 /usr/local/share/lua/5.1 下即可使用。

## 19.2 示例

下面给出几个示例, 分别演示了使用 luacurl 执行了 Get、Put、Upload 以及 MuLti-FTP Upload 操作。

### HTTP Get 示例:

```
local curl = require "lcurl"
local http = curl.easy{
    url = 'http://httpbin.org/get',
    httpheader = {
        "X-Test-Header1: Header-Data1",
        "X-Test-Header2: Header-Data2",
    },
    writefunction = io.stderr -- use io.stderr:write()
}
http:perform()
http:close()
http=nil
curl=nil
```

### HTTP Post 示例:

```
local curl = require "lcurl"
local http = curl.easy()
http:setopt_url('http://posttestserver.com/post.php')
http:setopt_writefunction(io.write)
http:setopt_httppost(curl.form()) -- Lua-cURL guarantee that form will be alive
http:add_content("test_content", "some data", {
    "MyHeader: SomeValue"
})
http:add_buffer("test_file", "filename", "text data", "text/plain", {
    "Description: my file description"
})
:add_file("test_file2", "BuildLog.htm", "application/octet-stream", {
    "Description: my file description"
```

```

    })
  )
  http:perform()
  http:close()

```

#### FTP Upload 示例:

```

local function get_bin_by(str,n)
  local pos = 1 - n
  return function()
    pos = pos + n
    return (str:sub(pos,pos+n-1))
  end
end

local curl = require "libcurl"
local ftp = curl.easy()
ftp:setopt_url("ftp://moteus:123456@127.0.0.1/test.dat")
ftp:setopt_upload(true)
ftp:setopt_readfunction(
  get_bin_by(("0123456789"):rep(4), 9)
)
ftp:perform()
ftp:close()

```

#### Multi FTP Upload 示例:

```

local curl = require "libcurl"
-- We get error E_LOGIN_DENIED for this operation
local e1 = curl.easy{url = "ftp://moteus:999999@127.0.0.1/test1.dat", upload =
true}
e1:setopt_readfunction(
  function(t) return table.remove(t) end, {"1111", "2222"}
)

local e2 = curl.easy{url = "ftp://moteus:123456@127.0.0.1/test2.dat", upload =
true}
e2:setopt_readfunction(get_bin_by(("e"):rep(1000), 5))

local m = curl.multi()
m:add_handle(e1)
m:add_handle(e2)

while m:perform() > 0 do m:wait() end

while true do
  h, ok, err = m:info_read()
  if h == 0 then break end

  if h == e1 then
    assert(ok == nil)
    assert(err:name() == "LOGIN_DENIED")
    assert(err:no() == curl.E_LOGIN_DENIED)
  end

  if h == e2 then

```

```

    assert(ok == true)
end
end

```

## 19.3 函数

luacurl 的函数用来创建封装了执行具体任务的对象。例如, form() 函数创建 HTTP 表单对象, easy() 函数创建简单的 curl 对象。具体的操作在生成的对象中进行。

### 1. form ()

功能: 创建 HTTP multipart/formdata 对象。

返回: (httpform) 新 curl HTTP Post 对象上下文。

### 2. easy ([options])

功能: 创建 easy 对象。

参数:

options: 表。

返回: easy 对象。

用法:

```

c = curl.easy{
    url = 'http://example.com',
    [curl.OPT_VERBOSE] = true,
}

```

### 3. multi ([options])

功能: 创建 multi 对象。

参数:

options: 表。

返回: multi 对象。

用法:

```

m = curl.multi{
    socketfunction = handle_socket;
    timerfunction = start_timeout;
}

```

### 4. share ([options])

功能: 创建 share 对象。

参数:

options: 表。

返回: share 对象。



### 5. version ()

功能：返回一个可读的 libcurl 库版本号。

### 6. version\_info ([param])

功能：以表返回 libcurl 版本号。

参数：

param：以字符串指定的版本信息。

返回：如果没有指定 param 则返回完整的版本信息，否则返回指定的信息。

用法：

```
proto = curl.version_info('protocols')
if proto.HTTP then ... -- libcurl support http protocol
```

## 19.3.1 httpform 类

httpform 是用于 POST 操作的表单对象，主要执行 POST 操作，携带 POST 所需要的数据。

### 1. httpform:add\_content (name, content[, type[, headers]])

功能：为 form 添加新部分。

参数：

name：新部分的名字。

content：实际发送的字符串型数据。

type：新部分以字符串描述的 Content-Type。

headers：以表为 POST 指定的扩展头域。

返回：self。

### 2. httpform:add\_buffer (name, filename, content[, type[, headers]])

功能：为 form 添加新的部分。

参数：

name：新部分的名字。

content：实际发送的字符串型数据。

type：新部分以字符串描述的 content-type。

headers：以表为 POST 指定的扩展头域。

filename：在 content 头域里的文件名。

返回：self (自身)。

### 3. httpform:add\_file (name, path[, type[, filename[, headers]]])

功能：为 form 添加新部分。

参数：

name: 新部分的名字。

path: 待发送的文件路径。

type: 新部分以字符串描述的 content-type。

headers: 以表为 POST 指定的扩展头域。

filename: 在 content 头域里的文件名。

返回: self。

#### 4. httpform:add\_stream (name[, filename][, type][, headers], size, reader[, context])

功能: 为 form 添加新部分。

参数:

name: 新部分的名字。

type: 新部分以字符串描述的 content-type。

headers: 以表为 POST 指定的扩展头域。

filename: 在 content 头域里的文件名。

size: 以字节为单位的流数量。

reader: function/object。

context: reader 的上下文。

返回: self。

#### 5. httpform:get ()

功能: 序列化 multipart/formdata HTTP POST 链。

返回: 字符串式序列化的数据。

用法:

```
print(post:get())
```

#### 6. httpform:get (writer[, context])

功能: 序列化 multipart/formdata HTTP POST 链。

writer 函数可以返回 true 或写完的数量。函数没有返回则也认为是成功的。

参数:

writer: 函数。

context: writer 上下文。

返回: self。

用法:

```
t = {}
post:get(table.insert, t)
print(table.concat(t))
```

### 7. httpform:get (writer)

功能：读取多部分 / 表单的 POST 数据。

本调用和 `httpform:get(writer.write, writer)` 一样。

参数：

`writer`：对象。

返回：`self`。

用法：

```
f = io.open(...)
post:get(f)
```

### 8. httpform:free ()

功能：释放 `multipart/formdata`。

## 19.3.2 easy 类

`easy` 类封装的是 HTTP 通用操作，可以执行相对较复杂的 GET 操作，但通常用来执行相对简单的操作。

#### 1. easy:perform ()

功能：执行一个文件传输。

返回：`self`。

#### 2. easy:escape (url)

功能：对 `url` 进行 URL 编码。

参数：

`url`：字符串。

返回：编码后的 URI。

#### 3. easy:unescape (url)

功能：对 `url` 进行 URL 解码。

参数：

`url`：字符串。

返回：解码后的 URI。

#### 4. easy:reset ()

功能：以之前的设置重新初始化。

返回：`easy self`。

#### 5. easy:pause (mask)

功能：暂停或恢复一个连接。

参数:

mask: 以位表示连接的新状态 (PAUSE\_XXX 常量)。

返回: easy self。

## 6. easy:close ()

功能: 结束 easy 会话。

## 7. easy:setopt (opt, ...)

功能: 设置操作。

参数:

opt: `curl.OPT\_XXX` 常量或配置表。

...: 值。

返回: easy self。

用法:

```
c:setopt(curl.OPT_URL, "http://example.com")
c:setopt(curl.OPT_READFUNCTION,
function(t, n) return table.remove(t) end,
{"1111", "2222"})
c:setopt{
  url = 'http://example.com',
  [curl.OPT_VERBOSE] = true,
}
```

## 8. easy:unsetopt (opt)

功能: 将操作重设为默认值。

参数:

opt: `curl.OPT\_XXX` 常量或配置表。

返回: easy self。

用法:

```
c:unsetopt(curl.OPT_URL)
c:unsetopt(curl.OPT_READFUNCTION)
```

## 9. easy:getinfo (info)

功能: 读取信息。

参数:

info: `curl.INFO\_XXX` 常量。

返回: 值。

用法:

```
print(c:getinfo(curl.INFO_EFFECTIVE_URL))
```

```
print(c:getinfo(curl.INFO_TOTAL_TIME))
print(c:getinfo(curl.INFO_RESPONSE_CODE))
```

### 10. easy:setopt\_writefunction (writer[, context])

功能：设置 writer 函数。

一个回调接受 1 或 2 个参数。第一个是 writer 的上下文（如果有），第二个是一个字符串式要写的数据。函数返回 true（任何非数值的 true 值）或完整的数据长度，否则传输将在任何错误下中断。

参数：

writer：函数

context：writer 上下文。

返回：self。

### 11. easy:setopt\_writefunction (writer)

功能：设置 writer 函数。与 easy:setopt\_writefunction(writer.write, writer) 相同。

参数：

writer：对象。

返回：self。

### 12. easy:setopt\_headerfunction (writer[, context])

功能：设置头函数。参见 easy:setopt\_writefunction (writer[, context])。

参数：

writer：对象。

context：writer 上下文。

返回：self。

### 13. easy:setopt\_headerfunction (writer)

功能：设置头函数。与 easy:setopt\_headerfunction(writer.header, writer) 相同。

参数：

writer：对象。

返回：self。

### 14. easy:setopt\_readfunction (reader[, context])

功能：设置 reader 函数。参见 easy:setopt\_writefunction (writer[, context])。

参数：

reader：函数。

context：reader 上下文。

返回：self。

用法：

```

local counter = 10
c:setopt_readfunction(function()
    if counter > 0 then
        counter = counter - 1
        return 'a'
    end
end)

```

### 15. easy:setopt\_readfunction (reader)

功能：设置 reader 函数。与 easy:setopt\_readfunction(reader.read, reader) 一样。

参数：

reader：对象。

返回：self。

### 16. easy:setopt\_progressfunction (progress[, context])

功能：设置 set progress 函数。

参数：

progress：函数。

context：progress 上下文。

返回：self。

### 17. easy:setopt\_progressfunction (progress)

功能：设置 progress 函数。

参数：

progress：对象。

返回：self。

### 18. easy:setopt\_httppost (data)

功能：设置 multipart/formdata。调用者不需要保存数据。

参数：

data：httpform。

返回：self。

### 19. easy:setopt\_postfields (data[, length=#data])

功能：设置 multipart/formdata。

参数：

data：字符串。

length：长度（默认 #data）。

返回：self。

## 20. easy:setopt\_share (data)

功能：设置 curl share 对象。调用者不需要保存数据。

参数：

data: share。

返回：self。

### 19.3.3 multi 类

multi 类执行并发操作，内部使用的是 easy 类，multi 类只是一个管理类。

#### 1. multi:add\_handle (handle)

功能：添加 easy 对象。

调用者必须确保 easy 对象是存活的，直到操作结束。

参数：

handle: easy。

返回：multi self。

#### 2. multi:remove\_handle (handle)

功能：移除 easy 对象。

参数：

handle: easy。

返回：multi self。

#### 3. multi:perform ()

功能：从每一个 easy 句柄中读 / 写有效数据。

返回：number (激活的 easy 句柄数量)。

#### 4. multi:info\_read ([remove])

功能：读取 multi 堆信息。

参数：

remove: 如果读取完成，移除 easy 句柄。

返回：如果没有信息，返回数值 0，或

(easy) 句柄

(boolean) true

或

(easy) 句柄

(nil)

(error) 错误代码

### 5. multi:setopt (opt, ...)

功能：设置操作。

参数：

opt: `curl.OPT\_MULTI\_XXX` 常量。

...: 值

返回：multi self。

用法：

```
c:setopt(curl.OPT_MULTI_MAXCONNECTS, 10)
c:setopt{maxconnects = 10}
```

### 6. multi:socket\_action ([socket=curl.SOCKET\_TIMEOUT[, mask=0]])

功能：执行套接字操作。

参数：

socket: 数值（默认 curl.SOCKET\_TIMEOUT）。

mask: 数值（默认 0）。

返回：multi self。

用法：

```
c:socket_action()
c:socket_action(sock_fd, curl.CSELECT_IN)
c:socket_action(sock_fd, curl.CSELECT_OUT)
```

### 7. multi:setopt\_timerfunction (timer[, context])

功能：设置时钟回调。

参数：

timer: 时钟回调。

context: 上下文。

返回：multi self。

### 8. multi:setopt\_timerfunction (timer)

功能：设置时钟回调。

参数：

timer: userdata 或表式的时钟对象。

返回：multi self。

### 9. multi:wait ([timeout])

功能：暂停一个 multi 对象中所有 easy 对象。

参数：

timeout: 以毫秒为单位的超时值，默认是 multi:timeout()。

返回：number（受影响的对象数）。



10. multi:timeout ()

功能：在处理之前的等待时间。

返回：number（以毫秒为单位的超时值）。

11. multi:close ()

功能：结束 multi 会话。

19.3.4 error 类

error 类是错误信息处理类。

1. error:category ()

功能：获取 error 类别。

返回：number [ 错误的类别号 (curl.ERROR\_XXX 常量)]。

用法：

```
if err:category() == curl.ERROR_EASY then
-- proceed easy error
end
```

2. error:no ()

功能：读取错误的数值值。

返回：number [ 错误的数值号 (curl.E\_XXX 常量)]。

3. error:name ()

功能：获取错误名字。

返回：string [ 错误名 (如 "UNSUPPORTED\_PROTOCOL"、"BAD\_OPTION")]

4. error:msg ()

功能：获取错误描述。

返回：string [ 错误描述 (如 "Login denied")]

5. error:\_tostring ()

功能：获取完整的错误描述。

返回：string (包含名字的字符串，如消息和错误值)。

19.3.5 share 类

share 类用于创建共享句柄。共享的类可以实现创建一次，然后分配给不同的对象使用的效果，以节省内存。easy 类支持设置共享句柄，有对应接口。

1. share:setopt (opt, ...)

功能：设置操作。

参数:

opt: `curl.OPT\_SHARE\_XXX` 常量的数值或表。

...: 值。

返回: share self。

用法:

```
c:setopt(curl.OPT_SHARE_SHARE, curl.LOCK_DATA_COOKIE)
c:setopt{share = curl.LOCK_DATA_COOKIE}
```

2. share:close ()

功能: 结束 share 会话。

19.4 常用变量

libcurl 中的常量非常多，这些量在 lcurl 中大部分都需要用到。本节将所有常用选项按类型分类列出，方便工作中使用。

19.4.1 字符串数组类选项

表 19-1 列出 lcurl 库常用字符串数组类常量，作为配置项时值应该是字符串数组。

表 19-1 lcurl 库常用字符串数组类常量

序号	选项	功能
1	curl.OPT_QUOTE	一组先于 FTP 请求的在服务器上执行的 FTP 命令
2	curl.OPT_POSTQUOTE	FTP 请求执行完成后，在服务器上执行的一组 FTP 命令
3	curl.OPT_TELNETOPTIONS	Telnet 选项
4	curl.OPT_PREQUOTE	传输之前执行一些命令
5	curl.OPT_HTTP200ALIASES	200 响应码数组，数组中的响应码被认为是正确的响应，否则被认为是错误的
6	curl.OPT_HTTPHEADER	HTTP 头域
7	curl.OPT_HTTPPOST	表单形式的 POST 数据部分

19.4.2 字符串选项

表 19-2 列出 lcurl 字符串类型常量，作为选项时值应该是字符串类型。

表 19-2 lcurl 字符串类型常量

序号	选项	功能
1	curl.OPT_FTP_ACCOUNT	发送 ACCT 命令
2	curl.OPT_URL	需要获取的 URL 地址
3	curl.OPT_PROXY	HTTP 代理通道
4	curl.OPT_USERPWD	连接中需要的用户名和密码，格式为 [username]:[password]
5	curl.OPT_PROXYUSERPWD	一个用来连接到代理的 "[username]:[password]" 格式的字符串

(续)

序号	选项	功能
6	curl.OPT_RANGE	以“X-Y”的形式,其中X和Y都是可选项,即获取数据的范围,以字节计。HTTP 传输线程也支持几个这样的重复项,中间用逗号分隔如“X-Y, N-M”
7	curl.OPT_POSTFIELDS	全部数据使用 HTTP 协议中的 POST 操作来发送。要发送文件,在文件名前面加上 @ 前缀并使用完整路径。这个参数可以通过 urlencoded 后的字符串类似 'para1=val1¶2=val2&...' 或使用以一个以字段名为键值,字段数据为值的数组。如果 value 是一个数组,Content-Type 头将会被设置成 multipart/form-data
8	curl.OPT_REFERER	在 HTTP 请求头中“Referer:”的内容
9	curl.OPT_FTPPORT	这个值将被用来获取供 FTP POST 指令所需要的 IP 地址。POST 指令告诉远程服务器连接到指定的 IP 地址。这个字符串可以是纯文本的 IP 地址、主机名、一个网络接口名 (UNIX 下) 或者只是一个“-”来使用默认的 IP 地址
10	curl.OPT_USERAGENT	在 HTTP 请求中包含一个“User-Agent:”头的字符串
11	curl.OPT_COOKIE	设定 HTTP 请求中“Cookie:”部分的内容。多个 cookie 用分号分隔,分号后带一个空格(例如,“fruit=apple; colour=red”)
12	curl.OPT_SSLCERT	包含 PEM 格式证书的文件名
13	curl.OPT_SSLKEYPASSWD	在 CURLOPT_SSLKEY 中指定了的 SSL 私钥的密码
14	curl.OPT_COOKIEFILE	包含 cookie 数据的文件名,cookie 文件的格式可以是 Netscape 格式,或者只是纯 HTTP 头部信息存入文件
15	curl.OPT_CUSTOMREQUEST	使用一个自定义的请求信息来代替 GET 或 HEAD 作为 HTTP 请求(对于执行 DELETE 或者其他更隐蔽的 HTTP 请求,有效值如 GET、POST、CONNECT 等等)。也就是说,不要在这里输入整个 HTTP 请求。例如,输入“GET /index.html HTTP/1.0\r\n\r\n”是不正确的。 注意:在确定服务器支持这个自定义请求的方法前不要使用
16	curl.OPT_WRITEINFO	写操作信息
17	curl.OPT_INTERFACE	网络发送接口名,可以是一个接口名、IP 地址或者一个主机名
18	curl.OPT_KRB4LEVEL	KRB4 (Kerberos 4) 安全级别。下面的任何值都是有效的(从低到高的顺序): clear、safe、confidential、private。如果字符串和这些都不匹配,将使用 private。这个选项设置为 NULL 时将禁用 KRB4 安全认证。目前 KRB4 安全认证只能用于 FTP 传输
19	curl.OPT_CAINFO	一个保存着 1 个或多个用来让服务端验证的证书的文件名。这个参数仅仅在和 CURLOPT_SSL_VERIFYPEER 一起使用时才有意义
20	curl.OPT_RANDOM_FILE	一个被用来生成 SSL 随机数种子的文件名
21	curl.OPT_EGDSOCKET	类似 CURLOPT_RANDOM_FILE,除了一个 Entropy Gathering Daemon 套接字
22	curl.OPT_COOKIEJAR	连接结束后保存 cookie 信息的文件
23	curl.OPT_SSL_CIPHER_LIST	一个 SSL 的加密算法列表。例如,RC4-SHA 和 TLSv1 都是可用的加密列表
24	curl.OPT_SSLCERTTYPE	证书的类型。支持的格式有 PEM (默认值)、DER 和 ENG
25	curl.OPT_SSLKEY	包含 SSL 私钥的文件名
26	curl.OPT_SSLKEYTYPE	CURLOPT_SSLKEY 中规定的私钥的加密类型,支持的密钥类型为 PEM (默认值)、DER 和 ENG

(续)

序号	选项	功能
27	curl.OPT_SSLENGINE	用来在 CURLOPT_SSLKEY 中指定的 SSL 私钥的加密引擎变量
28	curl.OPT_CAPATH	一个保存着多个 CA 证书的目录。这个选项是和 CURLOPT_SSL_VERIFYPEER 一起使用的
29	curl.OPT_ENCODING	HTTP 请求头中 “Accept-Encoding:” 的值。支持的编码有 identity、deflate 和 gzip。如果为空字符串，请求头会发送所有支持的编码类型

### 19.4.3 数值型选项

表 19-3 描述 libcurl 库数值型常量，作为选项时值必须是数值类型。

表 19-3 libcurl 库数值型常量

序号	选项	作用
1	curl.OPT_MAXREDIRS	指定最多的 HTTP 重定向的数量，这个选项是和 CURLOPT_FOLLOWLOCATION 一起使用的
2	curl.OPT_MAXCONNECTS	允许的最大连接数量，超过是会通过 CURLOPT_CLOSEPOLICY 决定应该停止哪些连接
3	curl.OPT_CLOSEPOLICY	关闭策略，不是 CURLCLOSEPOLICY_LEAST_RECENTLY_USED 就是 CURLCLOSEPOLICY_OLDEST
4	curl.OPT_CONNECTTIMEOUT	在发起连接前等待的时间，如果设置为 0，则无限等待
5	curl.OPT_SSL_VERIFYHOST	检查服务器 SSL 证书中是否存在一个公用名 (common name)。检查公用名是否存在，并且是否与提供的主机名匹配
6	curl.OPT_HTTP_VERSION	CURL_HTTP_VERSION_NONE (默认值，让 cURL 自己判断使用哪个版本)、CURL_HTTP_VERSION_1_0 (强制使用 HTTP/1.0) 或 CURL_HTTP_VERSION_1_1 (强制使用 HTTP/1.1)
7	curl.OPT_DNS_CACHE_TIMEOUT	设置在内存中保存 DNS 信息的时间，默认为 120 秒
8	curl.OPT_BUFFERSIZE	每次获取的数据中读入缓存的大小，但是不保证这个值每次都会被填满
9	curl.OPT_PROXYTYPE	不是 CURLPROXY_HTTP (默认值) 就是 CURLPROXY_SOCKS5
10	curl.OPT_HTTPAUTH	使用的 HTTP 验证方法，可选的值有 CURLAUTH_BASIC、CURLAUTH_DIGEST、CURLAUTH_GSSNEGOTIATE、CURLAUTH_NTLM、CURLAUTH_ANY 和 CURLAUTH_ANYSAFE。 可以使用   位域 (或) 操作符分隔多个值，cURL 让服务器选择一个支持最好的值。 CURLAUTH_ANY 等价于 CURLAUTH_BASIC   CURLAUTH_DIGEST   CURLAUTH_GSSNEGOTIATE   CURLAUTH_NTLM。 CURLAUTH_ANYSAFE 等价于 CURLAUTH_DIGEST   CURLAUTH_GSSNEGOTIATE   CURLAUTH_NTLM
11	curl.OPT_FTPSSLAUTH	FTP 验证方式: CURLFTPAUTH_SSL (首先尝试 SSL)、CURLFTPAUTH_TLS (首先尝试 TLS) 或 CURLFTPAUTH_DEFAULT (让 cURL 自动决定)
12	curl.OPT_POSTFIELDSIZE_LARGE	POST 数据以这个大小发送

(续)

序号	选项	作用
13	curl.OPT_PROXYAUTH	HTTP 代理连接的验证方式。使用在 CURLOPT_HTTPAUTH 中的位域标志来设置相应选项。对于代理验证只有 CURLAUTH_BASIC 和 CURLAUTH_NTLM 当前被支持
14	curl.OPT_FTP_RESPONSE_TIMEOUT	FTP 应答的超时值
15	curl.OPT_IPRESOLVE	IP 解析成的版本
16	curl.OPT_MAXFILESIZE	要去获得的最大文件尺寸
17	curl.OPT_INFILESIZE_LARGE	要发送的文件尺寸
18	curl.OPT_RESUME_FROM_LARGE	恢复一个传输器
19	curl.OPT_MAXFILESIZE_LARGE	要获取的最大文件尺寸
20	curl.OPT_PORT	用来指定连接端口
21	curl.OPT_TIMEOUT	设置 cURL 允许执行的最长时间（以秒为单位）
22	curl.OPT_INFILESIZE	设定上传文件的大小限制
23	curl.OPT_LOW_SPEED_LIMIT	最低速度限制 (bytes/sec)，低于则中断传输
24	curl.OPT_LOW_SPEED_TIME	达到最低速限制中断前的等待时间
25	curl.OPT_RESUME_FROM	在恢复传输时传递一个字节偏移量（用来断点续传）
26	curl.OPT_SSLVERSION	使用的 SSL 版本 (2 或 3)。默认情况下 PHP 会自己检测这个值，尽管有些情况下需要手动地进行设置
27	curl.OPT_TIMECONDITION	如果在 CURLOPT_TIMEVALUE 指定的某个时间以后被编辑过，则使用 CURL_TIMECOND_IFMODSINCE 返回页面。如果没有被修改过，并且 CURLOPT_HEADER 为 true，则返回一个 “304 Not Modified” 的 header；如果 CURLOPT_HEADER 为 false，则使用 CURL_TIMECOND_IFUNMODSINCE，默认值为 CURL_TIMECOND_IFUNMODSINCE
28	curl.OPT_TIMEVALUE	时间条件的值
29	curl.OPT_NETRC	在连接建立以后，访问 ~/.netrc 文件获取用户名和密码信息连接远程站点
30	curl.OPT_PROXYPORT	代理服务器的端口。端口也可以在 CURLOPT_PROXY 中进行设置
31	curl.OPT_POSTFIELDSIZE	POST 数据最大值

19.4.4 布尔型选项

表 19-4 列出 libcurl 库布尔型常量，作为选项时值必须是布尔型的。

表 19-4 libcurl 库布尔型常量

序号	选项	作用
1	curl.OPT_CRLF	将 UNIX 的换行符转换成回车换行符
2	curl.OPT_VERBOSE	汇报所有的信息，存放在 STDERR 或指定的 CURLOPT_STDERR 中
3	curl.OPT_HEADER	将头文件的信息作为数据流输出
4	curl.OPT_NOPROGRESS	关闭 CURL 传输的进度条，此项的默认设置为启用
5	curl.OPT_NOBODY	不对 HTML 中的 BODY 部分进行输出

(续)

序号	选项	作用
6	curl.OPT_FAILONERROR	显示 HTTP 状态码，默认行为是忽略编号小于等于 400 的 HTTP 信息
7	curl.OPT_UPLOAD	发起一个 HTTP UPLOAD 请求
8	curl.OPT_POST	发起一个 HTTP POST 请求
9	curl.OPT_FTPLISTONLY	列出 FTP 目录的名字
10	curl.OPT_FTPAPPEND	追加写入文件而不是覆盖它
11	curl.OPT_FOLLOWLOCATION	将服务器服务器返回的“Location:”放在 header 中递归地返回给服务器，使用 CURLOPT_MAXREDIRS 可以限定递归返回的数量
12	curl.OPT_TRANSFERTEXT	使用文本传输
13	curl.OPT_PUT	发起一个 HTTP PUT 请求
14	curl.OPT_AUTOREFERER	当根据 location 重定向时，自动设置 header 中的 Referer 信息
15	curl.OPT_HTTPPROXYTUNNEL	通过 HTTP 代理来传输
16	curl.OPT_TCP_NODELAY	TCP 的尼古拉算法
17	curl.OPT_FTP_CREATE_MISSING_DIRS	在远程服务器上创建丢失的目录
18	curl.OPT_UNRESTRICTED_AUTH	不在远程主机上限制校验
19	curl.OPT_FTP_USE_EPRT	FTP 下载时，使用 EPRT (或 LPRT) 命令。设置为 false 时禁用 EPRT 和 LPRT，使用 PORT 命令 only
20	curl.OPT_NOSIGNAL	忽略 signals 对应的处理函数
21	curl.OPT_COOKIESESSION	curl 仅仅传递一个 session cookie，忽略其他的 cookie，默认状况下 cURL 会将所有的 cookie 返回给服务端。session cookie 是指那些用来判断服务器端的 session 是否有效而存在的 cookie
22	curl.OPT_SSLENGINE_DEFAULT	默认的 SSL 引擎
23	curl.OPT_DNS_USE_GLOBAL_CACHE	启用一个全局的 DNS 缓存，此项为线程安全的，并且默认启用
24	curl.OPT_SSL_VERIFYPEER	校验 SSL 证书
25	curl.OPT_FILETIME	尝试修改远程文档中的信息。结果信息会通过 curl_getinfo() 函数的 CURLINFO_FILETIME 选项返回。curl_getinfo()
26	curl.OPT_FRESH_CONNECT	强制获取一个新的连接，替代缓存中的连接
27	curl.OPT_FORBID_REUSE	在完成交互以后强迫断开连接，不能重用
28	curl.OPT_FTP_USE_EPSV	FTP 传输过程中回复到 PASV 模式前首先尝试 EPSV 命令。设置为 false 时禁用 EPSV 命令
29	curl.OPT_HTTPGET	设置 HTTP 的 method 为 GET，因为 GET 是默认的，所以只在被修改的情况下使用

## 19.5 完整示例

下面是一个完整的使用示例，演示了最简单的 HTTP 操作。

```
local curl = require("lcurl")
```

```

local ipList =
{
    "192.168.1.1",
    "192.168.1.1",
}

-- 登录
function loginWeb(ip)
    c = curl.easy()
    c:setopt(curl.OPT_SSL_VERIFYHOST, 0);
    c:setopt(curl.OPT_SSL_VERIFYPEER, 0);
    c:setopt(curl.OPT_URL, "https://"..ip.."/")
    c:setopt(curl.OPT_POSTFIELDS, "Username=admin&Password=admin&Fr_m_
        Logintoken=&action=login")
    c:setopt(curl.OPT_WRITEFUNCTION, function(buffer)
        --print(buffer)
        --print("\r\n-----\r\n");
        return #buffer
    end
    )

    c:perform()
end

-- 访问页面
function accessPage(ip)
    c = curl.easy()
    c:setopt(curl.OPT_SSL_VERIFYHOST, 0);
    c:setopt(curl.OPT_SSL_VERIFYPEER, 0);
    c:setopt(curl.OPT_URL, "https://"..ip.."/xxpage.html")
    c:setopt(curl.OPT_WRITEFUNCTION, function(buffer)
        --print(buffer)
        --print("\r\n-----\r\n");
        return #buffer
    end
    )

    c:perform()
end

-- 设置参数
function setParameter(ip, file)
    c = curl.easy()
    c:setopt(curl.OPT_SSL_VERIFYHOST, 0);
    c:setopt(curl.OPT_SSL_VERIFYPEER, 0);
    c:setopt(curl.OPT_URL, "https://"..ip.."/xx.php")
    c:setopt(curl.OPT_POSTFIELDS, "DaylightSavingsUsed=1&Dscp=-1")

    local htmlTable = {}
    c:setopt(curl.OPT_WRITEFUNCTION, function(buffer)
        --print(buffer)
        --print("\r\n-----\r\n");
        table.insert(htmlTable, buffer)
        return #buffer
    end)

```



```

c:perform()

local htmlStr = table.concat(htmlTable);
local resultBuff = "";
if string.match(htmlStr, "<result>SUCC</result>") ~= nil then
    resultBuff = ip.." config OK\r\n";
    print(resultBuff)
    file:write(resultBuff);
else
    resultBuff = ip.." config NOK\r\n";
    print(resultBuff)
    file:write(resultBuff);
end
end

local file = io.open("./result.txt", "w+");for key,ip in ipairs(ipList) do
    loginWeb(ip);
    accessPage(ip);
    openLightSave(ip, file);
end
file:close();
print("Done")

```

## 19.6 小结

curl 库是一个强大的网络库，lcurl 是 Lua 内的 curl 库封装，让 Lua 拥有强大的网络功能。本章详细介绍了 curl 库支持的所有 curl 类和特性，为方便工作中查找 curl 常用变量，本章分类整理了 lcurl 支持的大部分常量。



## FFI 扩展 C 库

FFI 库是 LuaJIT 中最重要的一个扩展库，允许在纯 Lua 代码内调用扩展的 C 函数或使用 C 数据结构。FFI 库用于扩展 Lua 的功能，使 Lua 成为纯的“胶水”语言，可以整合丰富的 C/C++ 库，实现高性能的功能。使用 FFI 库可以从 C 工程头文件复制代码进 Lua 代码，把开发者从开发 Lua 扩展 C（语言 / 功能绑定库）的繁重工作中解放出来。

FFI 被紧密地整合进 LuaJIT 中，JIT 编译器为 Lua 代码直接访问 C 数据结构产生适配代码，等同于 C 编译器产生的代码。在 JIT 编译过的代码中，调用 C 函数被当作内联函数处理，不同于基于 Lua/C API 函数调用。

## 20.1 示例

下面通过两个示例介绍 FFI 的使用。

### 20.1.1 调用外部 C 函数

调用外部 C 函数是简单的，例如：

```
local ffi = require("ffi")
ffi.cdef[[
    int printf(const char *fmt, ...);
]]
ffi.C.printf("Hello %s!", "world")
```

具体分为 3 个步骤：

1) 载入 FFI 库。

2) 添加 C 函数的声明, 使用双 [] 号存放 C 定义。

3) 调用 C 函数。

步骤 3 中的 ffi.C 是标准库中的命名空间, 后面的 print 自动绑定到标准 C 库。传入的参数自动从 Lua 对象转换为相应的 C 类型。

上面的例子只是演示在纯 Lua 中调用 C 函数, 实际上可以使用 io.write() 和 string.format() 函数。下面的例子演示在 Windows 上弹出一个消息框:

```
local ffi = require("ffi")
ffi.cdef[[
    int MessageBoxA(void *w, const char *txt, const char *cap, int
type);
]]
ffi.C.MessageBoxA(nil, "Hello world!", "Test", 0)
```

可以使用这个机制使 Lua 和 C 协同工作: 创建一个外部的 C 文件, 添加 C 函数供 Lua 调用, Lua 传入参数, C 检查参数、校验密码。添加一个模块列表, 在 C 函数名前加上 luaopen\_\*, 注册所有的函数, 编译成一个共享连接库 (DLL), 放到合作的路径, 添加 Lua 代码载入模块, 最后调用函数。

### 20.1.2 使用 C 结构体数据

使用 FFI 库可以创建和访问 C 数据, 主要用途是为了和 C 函数接口, 为 C 接口封装所需要的 C 结构体数据作为参数, 当然, 也可以单独使用 C 的数据结构。

Lua 由高层数据结构构建, 这些数据结构复杂、可扩展、动态, 所以 Lua 被众多开发者喜爱。但是, 这些数据在某些任务下比较低效。例如, 要实现一个巨大的数组, 容纳一个大表, 其中包含许多小表, 这将需要一个大的内存负载和性能负载。

下例是一个操作颜色表的性能测试例子, 下面是纯 Lua 版本:

```
local floor = math.floor

local function image_ramp_green(n)
    local img = {}
    local f = 255/(n-1)
    for i=1,n do
        img[i] = { red = 0, green = floor((i-1)*f), blue = 0, alpha = 255 }
    end
    return img
end

local function image_to_grey(img, n)
    for i=1,n do
        local y = floor(0.3*img[i].red + 0.59*img[i].green + 0.11*img[i].blue)
        img[i].red = y; img[i].green = y; img[i].blue = y
    end
end

local N = 400*400
```

```

local img = image_ramp_green(N)
for i=1,1000 do
    image_to_grey(img, N)
end

```

本示例中创建了容纳 160 000 像素的表，每个像素是一个存放范围在 0 ~ 255 间数值的表。首先创建了一个绿色调色板，然后图像被转成灰度 1 000 次。

下面是 FFI 版本：

```

local ffi = require("ffi")
ffi.cdef[[
    typedef struct { uint8_t red, green, blue, alpha; } rgba_pixel;
]]

local function image_ramp_green(n)
    local img = ffi.new("rgba_pixel[?]", n)
    local f = 255/(n-1)
    for i=0,n-1 do
        img[i].green = i*f
        img[i].alpha = 255
    end
    return img
end

local function image_to_grey(img, n)
    for i=0,n-1 do
        local y = 0.3*img[i].red + 0.59*img[i].green + 0.11*img[i].blue
        img[i].red = y; img[i].green = y; img[i].blue = y
    end
end

local N = 400*400
local img = image_ramp_green(N)
for i=1,1000 do
    image_to_grey(img, N)
end

```

下面分析两者的不同点：

- 1) 载入 FFI 库，定义低层次的数据类型。这里可以选择使用一个 4B 的结构体，可以容纳 4 × 8 位 RGBA 像素。
- 2) 使用 ffi.new() 创建数据结构，[?] 是要创建的数组大小。
- 3) C 数组索引从 0 开始，所以索引是 0 ~ n-1。
- 4) ffi.new() 默认地对数组补零，只需要设置绿色和 alpha 域。
- 5) math.floor() 调用可以忽略，因为双浮点数在转换成整型时被截断了 0 部分。这是隐式地在数值存储进每一个像素域的时候进行的。

两份代码的不同点如下：

- 1) 内存消耗从 22MB 降到 640KB (400 × 400 × 4B)，降低了 35 倍。原来的纯 Lua 代码在 x64 平台上要消耗 40MB 内存。

2) Lua 代码运行消耗了 9.57 秒 (Lua 解释器消耗 52.9 秒), FFI 版本消耗了 0.48 秒, 提高了近 20 倍 (比 Lua 解释器约提高了 110 倍)。

## 20.2 FFI 库的使用

FFI 库定义了必需的类型转换函数, 以实现高级数据类型和 C 数据类型的转换, 同时提供了常用的支撑性方法。

### 20.2.1 载入 FFI 库

使用 require 载入 FFI 库。

```
local ffi = require("ffi")
```

不要在全局表中定义 ffi 变量, 需要使用局部变量。require 保证库只被载入一次。

### 20.2.2 访问标准系统函数

下面代码演示访问标准系统函数: 缓慢打印两行点号, 并在每个点后休眠 10 毫秒。

```
local ffi = require("ffi")
ffi.cdef[[
    void sleep(int ms);
    int poll(struct pollfd *fds, unsigned long nfd, int timeout);
]]

local sleep
if ffi.os == "Windows" then
    function sleep(s)
        ffi.C.sleep(s*1000)
    end
else
    function sleep(s)
        ffi.C.poll(nil, 0, s*1000)
    end
end

for i=1,160 do
    io.write("."); io.flush()
    sleep(0.01)
end
io.write("\n")
```

主要步骤解析:

1) 定义要使用的 C 函数, 定义在 [[]] 中。此处代码通常从 C 库头文件中获得, 或者从 C 库的文档中获得。

2) Windows 有 sleep() 函数, 与其他平台的 sleep() 函数有稍许不同, 一般是参数单位不同。其他平台使用 poll() 函数。ffi.os 用于判断平台, 以便调用正确的平台适应函数。

3) 在 Lua 函数中调用 C 函数, 在初始化时判断平台以避免每次调用时判断。

4) sleep() 参数的单位是秒, 所以将毫秒  $\times 1000$ 。传入的参数是 Lua 数值, 是一个双精度浮点数。方便的是 FFI 库在调用函数的时候自动进行转换。

Sleep() 是 KERNEL32.DLL 中的一个函数, 是一个 stdcall 函数。FFI 库提供了 ffi.C 这个默认的 C 库命名空间, 允许从默认库中调用函数, 像 C 编译器工作的原理一样。FFI 库自动检测 stdcall 函数, 所以会自动调用系统连接库中的函数。

5) poll() 函数调用需要几个参数, 可以使用 nil 传递空指针和 0 参数。注意, 0 不会转换为指针值, 和 C++ 不同, C++ 需要对各种类型参数提前赋值。

6) 在 Lua 代码里调用自己定义的 sleep() 函数,

## 20.2.3 访问 zlib 压缩库

下面代码演示从 Lua 代码中访问 zlib 压缩库。预先定义了两个转换接口, 携带一个字符串参数描述压缩和解压的数据。

```
local ffi = require("ffi")
ffi.cdef[[
    unsigned long compressBound(unsigned long sourceLen);
    int compress2(uint8_t *dest, unsigned long *destLen,
        const uint8_t *source, unsigned long sourceLen, int level);
    int uncompress(uint8_t *dest, unsigned long *destLen,
        const uint8_t *source, unsigned long sourceLen);
]]
local zlib = ffi.load(ffi.os == "Windows" and "zlib1" or "z")
local function compress(txt)
    local n = zlib.compressBound(#txt)
    local buf = ffi.new("uint8_t[?]", n)
    local buflen = ffi.new("unsigned long[1]", n)
    local res = zlib.compress2(buf, buflen, txt, #txt, 9)
    assert(res == 0)
    return ffi.string(buf, buflen[0])
end

local function uncompress(comp, n)
    local buf = ffi.new("uint8_t[?]", n)
    local buflen = ffi.new("unsigned long[1]", n)
    local res = zlib.uncompress(buf, buflen, comp, #comp)
    assert(res == 0)
    return ffi.string(buf, buflen[0])
end

-- Simple test code.
local txt = string.rep("abcd", 1000)
print("Uncompressed size: ", #txt)
local c = compress(txt)
print("Compressed size: ", #c)
local txt2 = uncompress(c, #txt)
assert(txt2 == txt)
```

操作步骤解析:

1) 定义 zlib 提供的 C 函数。

2) 载入 zlib 共享库 (在 POSIX 系统上称为 libz.so, 通常都会预装)。在 ffi.load() 自动添加或丢失标准前后缀, 我们只需要简单地载入 z 库。ffi.load() 中进行了平台判断, 传入正确的库名字。

3) 首先使用未压缩字符串长度调用 zlib.compressBound 获得压缩缓冲区。下行申请这个长度的缓冲区。? 指出数组变量长度。实际数组元素数量作为 ffi.new() 的第二个参数。

4) 目标长度被定义作为一个指针, 传入最大缓冲区长度并且传回实际使用长度。在 C 语言里, 需要传入本地变量的地址, 但是 Lua 中没有地址操作, 所以传入一个元素的数组。

5) 需要将压缩后数据作为一个 Lua 字符串返回, 所以使用 ffi.string()。本函数需要一个指针指向数据开始部分和实际长度。长度在 bufLen 数组中返回, 所以从数据中获取长度。

注意, 当函数返回后, buf 和 buflen 变量将被垃圾回收器回收。ffi.string() 已经将内容复制到一个新创建的 Lua 字符串中。如果计划多次调用这个函数, 考虑重用缓冲区, 把结果存放到缓冲区里, 这将降低垃圾回收的负载。

6) uncompress 函数的作用和 compress 函数是相反的。压缩数据不包含原始字符串的长度, 所以需要传入长度。

7) 使用刚定义的纯 Lua 代码函数, 这部分不需要熟悉 LuaJIT FFI。

注意 zlib API 使用 long 类型传送长度和尺寸。这些 zlib 函数实际只处理 32 位值。

long 类型在 POSIX/x64 系统上是 64 位, 但是在 Windows/x64 和 32 位系统上是 32 位。这样, 在 Lua 上, 一个 long 类型结果可以是一个 Lua32 位数值类型, 或一个 cdata 类型的 64 位数值, 依赖于具体的系统。所以, ffi.\* 函数接受 cdata 对象的时候, 无论是不是真正要使用 number, 都需要对返回结果使用 tonumber() 处理结果的 long 类型, 否则在很多系统上应用可能会失败。

## 20.2.4 为一个 C 类型定义元方法

下面代码为一个 C 类型定义了元方法, 定义了一个简单的 point 类型并且添加了一些操作。

```
local ffi = require("ffi")
ffi.cdef[[
    typedef struct { double x, y; } point_t;
]]

local point
local mt = {
    _add = function(a, b) return point(a.x+b.x, a.y+b.y) end,
    _len = function(a) return math.sqrt(a.x*a.x + a.y*a.y) end,
    _index = {
        area = function(a) return a.x*a.x + a.y*a.y end,
```

```
    },
}

point = ffi.metatype("point_t", mt)

local a = point(3, 4)
print(a.x, a.y)      --> 3  4
print(#a)             --> 5
print(a:area())       --> 25
local b = a + point(0.5, 8)
print(#b)             --> 12.5
```

主要步骤解析：

- 1) 使用 C 类型定义一个二维的 point 对象。
- 2) 首先定义变量存储 point 构造函数，因为需要在元方法之内使用。
- 3) 定义一个 \_add 元方法添加两个相关的 point 并且创建一个 point 对象。该函数假设两个参数是 point。但是可以是任意混合对象，如果至少一个运算符是请求的类型。\_len 元方法返回原点之间的距离。
- 4) \_index 表定义了一个 area 函数，还可以定义一个 \_newindex 函数代替。
- 5) point = ffi.metatype ("point\_t", mt) 只需要调用一次。ffi.metatype() 返回构造方式，我们不需要使用本方法。原始 C 类型可以创建一个 point 数组，原方法自动使用这些类型。
- 6) 这里是一些 point 类型和期待的结果。预定义的操作如 a.x 可以自由混合新定义的元方法。area 是必须通过 a:area() 调用的方法，而非 a.area()。

C 类型元方法是和面向对象风格 C 库连接时非常有用的机制，创建者返回一个指向新实例的指针，\_index 指向库命名空间，\_gc 指向完成的析构造函数。可以添加多个接口返回实际 Lua 字符串或返回多个值。

一些 C 库定义实例指针类型为 void \*。这种情况下，需要为所有声明定义一个假类型，如一个命名结构体的指针将这样做：typedef struct foo\_type \*foo\_handle。C 端代码不知道使用 LuaJIT FFI，但是因为底层类型的兼容性，可以正常工作。

20.2.5 转换 C 语法

表 20-1 列出通常的 C 语法与 LuaJIT FFI 对应关系。

表 20-1 C 语法与 LuaJIT FFI 对应关系

语法	C 代码	Lua 代码
Pointer dereference int *p;	x = *p; *p = y;	x = p[0] p[0] = y
Pointer indexing int i, *p;	x = p[i]; p[i+1] = y;	x = p[i] p[i+1] = y
Array indexing int i, a[];	x = a[i]; a[i+1] = y;	x = a[i] a[i+1] = y

(续)

语法	C 代码	Lua 代码
struct/union dereference struct foo s;	x = s.field; s.field = y;	x = s.field s.field = y
struct/union pointer deref. struct foo *sp;	x = sp->field; sp->field = y;	x = s.field s.field = y
Pointer arithmetic int i, *p;	x = p + i; y = p - i;	x = p + i y = p - i
Pointer difference int *p1, *p2;	x = p1 - p2;	x = p1 - p2
Array element pointer int i, a[];	x = &a[i];	x = a+i
Cast pointer to address int *p;	x = (intptr_t)p;	x = tonumber(ffi.cast("intptr_t", p))
Functions with outargs void foo(int *inoutlen);	int len = x; foo(&len); y = len;	local len = ffi.new("int[1]", x) foo(len)y = len[0]
Vararg conversions int printf(char *fmt, ...);	printf("%g", 1.0); printf("%d", 1);	printf("%g", 1); printf("%d", ffi.new("int", 1))

20.3 FFI API

FFI 提供了 7 类 API，涉及类型转换、功能操作、类型信息、标准库扩展等。

20.3.1 声明和访问外部符号

外部符号必须首先声明并且可以被一个库命名空间索引和访问，库自动绑定符号。

1. ffi.cdef(def)

ffi.cdef(def) 用于添加多个 C 类型或扩展符号（命名变量或函数）。def 必须是一个 Lua 字符串。例如：

```
ffi.cdef[[
typedef struct foo { int a, b; } foo_t; // Declare a struct and typedef.
int dofoo(foo_t *f, int n); /* Declare an external C function. */
]]
```

字符串的内容必须是一系列 C 声明，使用分号分隔。为避免错误，推荐严格遵守 C 语法。扩展符号只是声明，没有绑定到任何地址，绑定由 C 库命名空间实现。

2. ffi.C

ffi.C 是默认的 C 库命名空间，绑定到目标系统的符号或库。

3. clib = ffi.load(name [,global])

装载 name 的动态库，返回绑定到符号的 C 库命名空间。在 POSIX 系统上，如果 global 设置为 true，库符号将装载入全局命名空间。



如果 name 携带路径, 库将从目录中载入, 否则将从系统默认查找路径查找。

在 POSIX 系统上, 如果名字不包含 “.”, 将自动添加 .so 为后缀, 系统也会自动添加前缀, 所以 ffi.load(“z”) 在实际的 POSIX 系统上执行结果为: 在共享库目录中查找 libz.so。

### 20.3.2 创建 cdata 对象

下列 API 函数用于创建 cdata 对象, 所有创建的 cdata 对象都被垃圾回收器管理。

1. `cdata = ffi.new(ct [,nelem] [,init...])`, `cdata = ctype([nelem,] [init...])`

说明: 用 ct 创建 cdata 对象, VLA/VLS 类型需要 nelem 参数。第二个使用 ctype 作为构造器, 其他都是一样的。

cdata 对象依照初始化规则初始化, 使用可选的 init 参数。

如果要创建相同类型的多个对象, 只要解析 cdecl 一次, 通过 ffi.typeof() 获取它的 ctype, 然后使用 ctype 重复创建。

2. `ctype = ffi.typeof(ct)`

说明: 创建 ct 的 ctype 对象。这个函数用于解析 cdecl 一次, 使用结果 ctype 对象作为一个构造器。

3. `cdata = ffi.cast(ct, init)`

说明: 使用 ct 创建一个标量 cdata 对象, cdata 对象可以使用 init 初始化。

4. `ctype = ffi.metatype(ct, metatable)`

说明: 使用 ct 和绑定的元表创建一个 ctype 对象。只允许 struct/union 类型、vector 和复数, 其他类型可以使用 struct 封装。

元表需要持久并且在未来不能被改变, 元表的内容和 \_index 表的内容都不能被编辑。

5. `cdata = ffi.gc(cdata, finalizer)`

说明: 关联一个 finalizer 到 cdata 对象, 这个对象被无改变返回。

这个函数允许安全地将一个 cdata 对象绑定到 LuaJIT 垃圾回收器上。

```
local p = ffi.gc(ffi.C.malloc(n), ffi.C.free)
```

```
...
```

```
p = nil -- 最后一个 p 调用
```

```
-- GC 将调用 finalizer: ffi.C.free(p)
```

finalizer 可以是 Lua 函数或是一个 cdata 函数或是一个 cdata 函数指针。可以通过设置 nil 移除一个 finalizer。

```
ffi.C.free(ffi.gc(p, nil)) -- 手动释放内存
```

### 20.3.3 C 类型信息

下列 API 函数返回 C 类型信息, 这对检验 cdata 对象非常有用。

1. `size = ffi.sizeof(ct [,nelem])`

说明：返回 `ct` 尺寸（以字节为单位）。如果尺寸未知，则返回 `nil`（如 `void` 类型）。VLA/VLS 类型需要 `nelem` 参数。

2. `align = ffi.alignof(ct)`

说明：返回 `ct` 最小对齐需求（字节为单位）。

3. `ofs [,bpos,bsize] = ffi.offsetof(ct, field)`

说明：返回 `ct` 在 `field` 中的偏移量，`field` 必须是结构体。

4. `status = ffi.istype(ct, obj)`

说明：如果 `obj` 是 `ct` 的 C 类型，则返回 `true`，否则返回 `false`。

C 类型修改辞（`const` 等）被忽略，指针按标准指针兼容规则检查，但是对 `void*` 不做任何特殊对待。如果 `ct` 指定是 `struct/union`，将接受一个指针，否则必须精确匹配。

## 20.3.4 功能函数

下面是 FFI 库功能函数，用于对返回库状态或执行库级支撑功能。

1. `err = ffi.errno([newerr])`

说明：返回上次 C 函数设置的错误码。如果给定可选的 `newerr` 参数，将返回前一个错误码并将当前错误码设置成新的值。

函数提供的是平台无关性的错误码，注意，只有部分的 C 函数设置错误码，但并不标示实际错误条件（如返回 `-1` 或 `null`），而且可能包含上次设置的值。

2. `str = ffi.string(ptr [,len])`

说明：从 `ptr` 数据指针创建并保留一个 Lua 字符串。

如果没有给出可选的 `len` 参数，`ptr` 转成一个 `char*` 类型，数据用 0 终结，长度使用 `strlen()` 计算。

如果给出 `len` 值，`ptr` 转换成 `void*` 类型，`len` 表示数据的长度。

这个函数用于将 C 函数返回的 `const char*` 指针转换为 Lua 字符串并且存储数据传输到其他函数。Lua 字符串是 8 位单字节，可以用来保存任意非字符串数据。

注意：给 `len` 赋值会使本函数比不给 `len` 赋值更快。

3. `ffi.copy(dst, src, len), ffi.copy(dst, str)`

说明：将数据从 `src` 复制到 `dst`。`dst` 被转换为 `void*`，`src` 被转换为 `const void*`。

第一个表达式中，`len` 用于指定要复制的长度。如果 `src` 是一个 Lua 字符串，`len` 必须不能超过 `#src+1`。

第二个表达式中，源数据必须是一个 Lua 字符串，字符串所有的字节包括终结符 0 都会被复制到 `dst`（`#src+1` 字节）。

注意：`ffi.copy` 可以用 C 库中类似于 `memcpy()`、`strcpy()` 和 `strncpy()` 等更快的函数替换。

#### 4. ffi.fill(dst, len [,c])

说明：dst 中的 len 个数据使用 c 填充，如果没有 c 参数，使用 0 填充。

注意：ffi.fill() 可以使用 C 库中的 memset(dst, c, len) 替换。

### 20.3.5 特定目标信息

本节函数用于返回目标主机的特定信息，一般用于平台信息获取。

#### 1. status = ffi.abi(param)

说明：如果 param (Lua 字符串) 被 ABI (Application Binary Interface) 理解，则返回 true，否则返回 false。表 20-2 列出 param 支持的参数。

表 20-2 param 支持的参数

参数	描述	参数	描述
32bit	32 位架构	softfp	在 softfp 模式下使用 FPU 做浮点运算
64bit	64 位架构	hardfp	在 hardfp 模式下使用 FPU 做浮点运算
le	小头结构	eabi	标准 ABI 的 EABI 变量
be	大头结构	win	标准 ABI 的 Windows 变量
fpu	目标机有硬件 FPU		

#### 2. ffi.os

说明：包含目标 OS 名称，内容和 jit.os 内容相同。

#### 3. ffi.arch

说明：包含目标架构名字，内容和 jit.arch 相同。

### 20.3.6 方法回调

库提供了两个用于回调函数的操作。

#### 1. cb:free()

说明：释放回调绑定的资源。绑定的 Lua 函数是固定的并且可以被回收。操作后，回调函数指针不再有效并且不能再被调用，可以被重用。

#### 2. cb:set(func)

说明：将 Lua 函数和回调关联起来。但是 C 类型的回调和回调函数指针都不会改变。

这个函数用于动态切换回调的接收器，而不需要每次重新创建一个新的回调并重新注册。

### 20.3.7 扩展标准库函数

下列标准库函数扩展和 cdata 对象共同工作。

#### 1. n = tonumber(cdata)

说明：转换 cdata 对象为 double 型，并且返回 Lua 数值。本函数对 64 位整数封装特别

有用。

## 2. s = tostring(cdata)

说明：返回描述 64 位整数的字符串 ("nnnLL" 或 "nnnULL") 或复数 ("re ± imi")。如果 cdata 不是整数类型，则返回一个描述 ctype 的对象 ("ctype<type>") 或一个 cdata 对象 ("cdata<type>: address")。注：如果使用 \_tostring 元方法覆盖了方法，则会使元方法操作 cdata。

## 3. iter, obj, start = pairs(cdata), iter, obj, start = ipairs(cdata)

说明：调用 \_pairs 或 \_ipairs 元方法。

## 20.4 调用 curl 库的完整示例

下面是一个调用 curl 的 FFI 示例，这个示例演示了直接使用 C 库或使用封装库（可以直接使用 lcurl 库）的差别。

```
local ffi = require 'ffi'

ffi.cdef[[
    void *curl_easy_init();
    int curl_easy_setopt(void *curl, int option, ...);
    int curl_easy_perform(void *curl);
    void curl_easy_cleanup(void *curl);
    char *curl_easy_strerror(int code);
]]

local libcurl = ffi.load('libcurl')

local curl = libcurl.curl_easy_init()
local CURLOPT_URL = 10002 -- 参考 curl/curl.h 中定义

if curl then
    libcurl.curl_easy_setopt(curl, CURLOPT_URL, 'http://example.com')
    res = libcurl.curl_easy_perform(curl)
    if res ~= 0 then
        print(ffi.string(libcurl.curl_easy_strerror(res)))
    end
    libcurl.curl_easy_cleanup(curl)
end
```

## 20.5 小结

FFI 库是 LuaJit 中的一个重要库，使用 FFI 库可以让 Lua 调用 C/C++ 的代码和库，让 Lua 拥有了丰富的资源。本文详细介绍了 FFI 库的使用方法，并详细描述了一个 FFI API。通过本章的学习，读者可以在自己的 Lua 程序中使用 C 函数和数据结构。

## cjson 库的使用

Lua cjson 库为 Lua 提供了 JSON 处理能力，是一个快速的 JSON 处理库，包含在 OpenResty 内，可以使用编译开关打开或关闭，默认是打开的。

Lua cjson 模块特点如下：

- 快速，支持标准的编解码操作。
- 完全支持 UTF-8。
- 可选运行期 JSON 异常支持。
- 不支持 UTF-16 和 UTF-32。

### 21.1 示例

示例：演示 cjson 库使用方法，cjson 库使用还是相对简洁的。

```
-- 模块初始化
local cjson = require "cjson"
local cjson2 = cjson.new()
local cjson_safe = require "cjson.safe"

-- 将 Lua 值转换为 JSON
text = cjson.encode(value)
value = cjson.decode(text)

-- 读 / 写 CJSON 配置
setting = cjson.decode_invalid_numbers([setting])
setting = cjson.encode_invalid_numbers([setting])
keep = cjson.encode_keep_buffer([keep])
depth = cjson.encode_max_depth([depth])
```

```
depth = cJSON.decode_max_depth([depth])
convert, ratio, safe = cJSON.encode_sparse_array([convert[, ratio[, safe]]])
```

## 21.2 函数

CJSON 的函数不多，常用的是用于编码的 `encode` 和用于解码的 `decode`。

### 1. 模块导入

```
local cJSON = require "cjson"
local cJSON2 = cJSON.new()
local cJSON_safe = require "cjson.safe"
```

通过 `require` 函数导入 `cjson` 库，`cjson` 库不注册全局模块表。

当 `cjson` 模块遇到无效数据时，将抛出错误。参见 `cjson.encode` 和 `cjson.decode` 查看细节。

`cjson.safe` 模块和 `cjson` 模块一致，除了在 JSON 转换过程中对错误的处理方式不同。

错误发生时，`cjson_safe.encode` 和 `cjson_safe.decode` 函数将返回 `nil` 和错误描述信息。

`cjson.new` 可以用于实例一个独立的 `cjson` 模块。新模块拥有一个独立编码缓冲区和默认设置。

Lua `cjson` 可以在单 Lua 提供一个不共享的持续缓冲区的情况下使 Lua 代码使用多个优先线程，可以被下列方法实现。

- 使用 `cjson.encode_keep_buffer` 使缓冲区无效。
- 确保每一个线程分别调用 `cjson.encode`。
- 在优先线程中使用分隔的 `cjson` 模块表 (`cjson.new`)。

注意：CJSON 使用 `strtod` 和 `sprintf` 实现数值转换。然而，这些函数需要一个工作区用于编码/解析。CJSON 在导入时候自动侦测当前区域，在需要的时候自动实现工作区。如果当前进程区域改变了，需要通过 `cjson.new` 重新初始化 CJSON。CJSON 不支持在一个线程中使用不同的区域。

### 2. decode

语法：

```
value = cJSON.decode(json_text)
```

`cjson.decode` 反序列化任意的 UTF-8 JSON 字符串为 Lua 值或表。不支持 UTF-16 和 UTF-32 字符串。

`cjson.decode` 需要 NULL (ASCII 0) 和双引号 (ASCII 34) 分离内嵌的字符串。转义码将被解码，其他的字节将被透明传输。UTF-8 字符只在需要的时候检查。

JSON null 被转成 NULL 值，可以使用 `cjson.null` 进行比较。

默认地，数值不和 JSON 规格兼容（无穷大，NaN，十六进制）的可以解码，这个默认值可以通过 `cjson.decode_invalid_numbers` 修改。

解码示例:

```
json_text = '[ true, { "foo": "bar" } ]'
value = cJSON.decode(json_text)
-- Returns: { true, { foo = "bar" } }
```

注意: 以数值作为 key, CJJSON 都将返回字符串, 所以不能直接以数值进行 key 判断。

### 3. decode\_invalid\_numbers

语法:

```
setting = cJSON.decode_invalid_numbers([setting])
```

该指令用于设置无效的数值类型, setting 必须是布尔型, 默认是 true。

当解码 JSON 规格中不支持的数值时, 将会产生一个错误。无效的数值如下:

- 无穷大。
- 非数值 (NaN)。
- 十六进制。

有效的 setting 值:

- true: 接收并解码无效数值, 这是默认设置。
- false: 当遇到无效数值时, 抛出一个错误。

### 4. decode\_max\_depth

语法:

```
depth = cJSON.decode_max_depth([depth])
```

该指令用于解析最大深度。当数组 / 对象解析深度达到上限时, 将产生一个错误, 用于防止混乱的 JSON 格式拖慢应用程序, 或者由此引起的堆栈溢出。

depth 必须是正整数, 默认为 1000。

### 5. encode

语法:

```
json_text = cJSON.encode(value)
```

cJSON.encode 将 Lua 值编码进一个字符串。

cJSON.encode 支持下面类型:

- boolean;
- lightuserdata (空值);
- nil;
- number;
- string;
- table。

下面的 Lua 类型将引发错误:

- function;
- lightuserdata (非空值);
- thread;
- userdata。

默认地, 数值编码为 14 个数字, 可在 `cjson.encode_number_precision` 查看细节。

Lua CJSON 将 UTF-8 字符串中的下列字符进行编码:

- 控制字符 (ASCII 0 ~ 31);
- 双引号 (ASCII 34);
- 正斜杠 (ASCII 47);
- 黑斜线 (ASCII 92);
- 删除键 (ASCII 127)。

所有其他字节直接传输。

注意: Lua CJSON 可以成功编码 / 解码二进制串, 但这是 JSON 不支持的, 也与其他 JSON 库不兼容。保证输出是有效的 JSON, 应用程序需要确保所有传入 `cjson.encode` 的字符串是 UTF-8。通常使用 Base64 对二进制编码, 然后嵌入 UTF-8 串。Lua Base64 可以在 `LuaSocket` 和 `Base64` 包中找到。

Lua CJSON 试探决定将一个 Lua 表编码为 JSON 数组或一个对象。表中的 key 是数值型, 将编码为一个 JSON 数组, 其他的表将编码为 JSON 对象。

Lua CJSON 序列化表时, 不使用元方法。

- `rawget` 用于迭代 Lua 数组。
- `next` 用于迭代 Lua 对象。

JSON 对象 keys 总是字符串, 因此 `cjson.encode` 支持表 keys 是字符串和数值类型, 对其他类型会产生错误。

注意: 标准 JSON 串必须装入 `{}` 或 `[]`。表必须通过 `cjson.encode` 编码。

默认, 编码下列 Lua 值将产生错误:

- 不兼容的数值类型 (infinity, NaN);
- 超过 1000 层的表;
- 过度稀疏 Lua 数组。

这些默认值可以通过下列方法修改:

- `cjson.encode_invalid_numbers`;
- `cjson.encode_max_depth`;
- `cjson.encode_sparse_array`。

编码示例:



```
value = { true, { foo = "bar" } }
json_text = cJSON.encode(value)-- Returns: '[true,{"foo":"bar"}]'
```

## 6. encode\_invalid\_numbers

语法:

```
setting = cJSON.encode_invalid_numbers([setting])
```

该指令用于设置无效值的处理方法。setting 必须是布尔值或 null，默认是 false。

当 Lua CJSON 编码 JSON 规格不支持的数据类型时会产生错误:

- 无穷大;
- 非数值 (NaN)。

setting 值:

- true: 允许无效数值编码。将产生一个不标准的 JSON，但有些库支持这种输出。
- null: 编码无效数值为一个 JSON null 值。允许无穷大和 NaN 编码为有效的 JSON。
- false: 当遇到无效数值时，抛出错误。这是默认值。

## 7. encode\_keep\_buffer

语法:

```
keep = cJSON.encode_keep_buffer([keep])
-- "keep" must be a boolean. Default: true.
```

Lua CJSON 可以重用编码缓冲区以改善性能。keep 必须是布尔值，默认是 true。

keep 值:

- true: 默认值，缓冲区将增长到需要的最大值，并且直到 cJSON 模块被回收。
- false: 在每一次 cJSON.encode 调用完释放编码缓冲区。

## 8. encode\_max\_depth

语法:

```
depth = cJSON.encode_max_depth([depth])
```

该指令用于设置编码最大表深度，depth 必须是正整数，默认为 1000。

## 9. encode\_number\_precision

语法:

```
precision = cJSON.encode_number_precision([precision])
```

该指令用于修改数值型编码为文件的字节数，用于改善性能。例如，若 precision 改为 3，则可以提升编码性能到 50%。precision 必须是 1 ~ 14 间的一个数值，默认是 14 字节。

## 10. encode\_sparse\_array

语法:

```
convert, ratio, safe = cJSON.encode_sparse_array([convert[, ratio[, safe]]])
```

Lua CJSON 将 Lua 表编码为 JSON 三种数组中的一种，由 Lua 数组中丢失的值数量决定：

- 普通：所有值有效。
- 稀疏：至少 1 个值丢失。
- 过度稀疏：丢失的值超过设置的比例。

Lua CJSON 编码稀疏数组时，使用 JSON null 作为丢失的条目。

当满足下列条件时，一个数组判定为稀疏数组：

- `ratio > 0`;
- `maximum_index > safe`;
- `maximum_index > item_count * ratio`。

3 个参数如下：

- `convert`：必须是布尔值，默认为 `false`。
- `ratio`：必须是正整数，默认为 2。
- `safe`：必须是正整数，默认为 10。

Lua CJSON 不会在 `ratio=0` 时认为一个数组是稀疏数组。`safe` 限制保证小的数组总是编码为稀疏数组。

默认，企图编码一个绝对稀疏数组将产生一个错误，如果 `convert` 设置为 `true`，绝对稀疏数组将转换为 JSON 对象。

例如，编码一个稀疏数组：

```
cjson.encode({ [3] = "data" })
-- Returns: '[null,null,"data"]'
```

例如，使能转换为 JSON 对象：

```
cjson.encode_sparse_array(true)
cjson.encode({ [1000] = "excessively sparse" })
-- Returns: '{"1000":"excessively sparse}"'
```

## 21.3 变量

CJSON 提供了 3 个变量供使用。

- 1) `_NAME`：返回 CJSON 名字（“cjson”）。
- 2) `_VERSION`：返回 cjson 模块版本号（“2.1.0”）。
- 3) `null`：Lua CJSON 解码 JSON null 为一个用户级 NULL 指针，`cjson.null` 用于兼容性。

## 21.4 小结

Web 开发中广泛使用 JSON 数据格式，所以 `cjson` 库是 Lua 中的一个重要基础库。本章介绍了 CJSON 的用法和实例。因为 JSON 本身简洁易读，所以 `cjson` 库使用也非常简单，容易上手。

## lua-resty-template 类的使用

Web 开发中经常使用到动态 Web 网页开发技术，如淘宝商品页，详情页面显示非常复杂，逻辑也非常复杂，一般都是使用动态页面技术实现的，常见的 Web 端动态页面技术是 PHP、JSP 等。但在商品详情这种页面中变动又不是那么快速，使用 PHP、JSP 等又不完全合适，需要组建复杂的系统，使用 CGI 缓存等技术提高系统整体并发能力，这时可以使用模板技术实现。Lua 中有许多模板引擎，这里介绍 OpenResty 团队提供的 lua-resty-template，可以渲染很复杂的页面，在 LuaJIT 上的性能表现不错。

模板的工作机理与 JSP 类似。页面是模板，中间有定义好的标签。页面被提交到后端后，Servlet（Servlet 是服务端程序）对模板进行翻译，将变量翻译成系统动态的数据，然后返回给客户端。lua-resty-template 模板也是一样的机制，模板页面被 template 模块翻译成 Lua 模板，然后通过 ngx.print 输出。

lua-resty-template 提供了下面的工作能力：

- 模板位置：模板存放位置。
- 变量输出 / 转义：变量值输出。
- 代码片段：执行代码片段，完成如 if/else、for 等复杂逻辑，调用对象函数 / 方法。
- 注释：解释代码片段含义。
- include：包含另一个模板片段。
- 其他：不需要解析片段、简单布局、可复用的代码块、宏指令等。

### 22.1 示例

示例：通过模板输出“Hello World!”。

Lua 代码如下:

```
local template = require "resty.template"
view = template.new "view.html"
view.message = "Hello, World!"
view:render()
template.render("view.html", { message = "Hello, World!" })
```

view.html 内容如下:

```
<!DOCTYPE html>
<html>
<body>
<h1>{message}</h1>
</body>
</html>
```

输出如下:

```
<!DOCTYPE html>
<html>
<body>
<h1>Hello, World!</h1>
</body>
</html>
```

Lua 代码可以存放到 nginx.conf 相应的 location 里, 实现访问指定 location 输出格式化内容。

也可以使用内联模板字符串实现:

```
-- 使用内联模板字符串
template.render([[<!DOCTYPE html><html><body><h1>{message}</h1></body></html>]], { message = "Hello, World!" })
```

## 22.2 模板符号

现在模板中定义了下面的标签。

- `{{expression}}`: 表达式的结果, HTML 转义。
- `{*expression*}`: 表达式的结果。
- `{% lua code %}`: 运行 Lua 代码。
- `{{(template)}}`: 包含模板文件, 也可以为包含文件提供内容, 如 `{{(file.html, { message = "Hello, World" } )}}`。
- `[[expression]]`: 包含表达式文件。同样可以为文件提供内容 (表达式的结果), 如 `[[ "file.html", { message = "Hello, World" } ]]`。
- `{-block-}...{-block-}{-block-}`: 一个以 block 为 key 的一个 blocks 表的值。这种情况

不能使用预定义块。

- `{-verbatim-}...{-verbatim-}` 和 `{-raw-}...{-raw-}`：预定义块，不被模板模块运行，但是会输出。
- `{# comments #}`：注释 # 中间的是注释内容，不会输出和运行。

使用模板可以访问上下文表以及模板表中的所有内容，也可以使用前缀访问，例如：

```
<h1>{{message}}</h1> == <h1>{{context.message}}</h1>
```

## 22.2.1 短转义符号

如果不想输出一个模板符号，在标签开头加上反斜杠 \：

```
<h1>\{{message}}</h1>
```

将输出：

```
<h1>{{message}}</h1>
```

如果想在模板标签前输出反斜杠，需要将反斜杠转义：

```
<h1>\\{{message}}</h1>
```

将输出：

```
<h1>[message-variables-content-here]</h1>
```

## 22.2.2 上下文表中的复杂 key

可能会有这种类型的上下文表：

```
local ctx = {"foo:bar" = "foobar"}
```

想在模板中渲染 `ctx["foo:bar"]` 的值的时候，可以在模板中这样引用：

```
{# {*[ "foo:bar" ]*} won't work, you need to use: #}  
{*context["foo:bar"]*}
```

或者：

```
template.render([[{*context["foo:bar"]*}]], {"foo:bar" = "foobar"})
```

## 22.2.3 HTML 转义

只有字符串会被转义，没有参数的函数调用返回结果会被转义，其他类型要使用 `tostring`，`nil` 和 `ngx.nulls` 被转成 “”。

转义的 HTML 字符如下：

- `& ->`      `&amp;`;
- `< ->`      `&lt;`;
- `> ->`      `&gt;`;

- “ ->     "
- ‘ ->     #39;
- / ->     #47;

例如:

```
--Lua 代码
local template = require "resty.template"
template.render("view.html", {
    title   = "Testing lua-resty-template",
    message = "Hello, World!",
    names   = { "James", "Jack", "Anne" },
    jquery  = '<script src="js/jquery.min.js"></script>'
})
```

view.html 内容如下:

```
{{header.html}}
<h1>{{message}}</h1>
<ul>
{% for _, name in ipairs(names) do %}
<li>{{name}}</li>
{% end %}
</ul>
{{footer.html}}
```

header.html 内容如下:

```
<!DOCTYPE html>
<html>
<head>
<title>{{title}}</title>
    {{*jquery*}}
</head>
<body>
```

footer.html 内容如下:

```
</body>
</html>
```

## 22.2.4 保留的上下文 key 和评论

下面的 key 不要在上下文表中使用:

- `_`: 存放编译模板, 如果设置了内容可以通过 `{{context._}}` 使用。
- `context`: 存放当前上下文, 如果设置了内容可以通过 `{{context.context}}` 使用。
- `include`: 保存 include 的 helper 函数, 通过 `{{context.include}}` 使用。
- `layout`: 存储 view 中要使用的 layout, 通过 `{{context.layout}}` 使用。
- `blocks`: 存储 blocks, 通过 `{{context.blocks}}` 使用。
- `template`: 存储模板表, 通过 `{{context.template}}` 使用。

template.new 也不要被覆盖:

- render: 用来渲染一个 view。

不能有 `{{(view.html)}}` 递归调用, 例如:

```
Lua
template.render "view.html"
```

```
view.html
{{(view.html)}}
```

解决这个问题可以通过使用 `{{( 符号 )}}` 从子路径载入模板:

```
view.html
{{(users/list.html)}}
```

同样, 可以通过文件路径或一个字符串提供模板, 如果文件存在, 将使用文件, 否则当成一个字符串使用。

## 22.3 安装

只需要把 `template` 路径和 `template.lua` 文件复制到系统的 `package.path` 路径下即可, 在 `resty` 路径下。如果使用 `OpenResty`, 默认位置是 `/usr/local/openresty/lualib/resty`。

下载源文件:

```
git clone https://github.com/bungle/lua-resty-template
```

安装:

```
cd lua-resty-template/lib
cp -r resty /usr/local/openresty/lualib/
```

### 22.3.1 Nginx/OpenResty 配置

`lua-resty-template` 在 `Nginx/ OpenResty` 上使用, 需要在 `nginx.conf` 的 `server` 部分进行一些配置:

- `template_root(set $template_root /var/www/site/templates;)`。
- `template_location(set $template_location /templates;)`。

如果在 `nginx.conf` 中配置了这两个问题, 则可以使用 `ngx.var.document_root`。设置 `template_location` 后, 需要测试一下, 以查看对应的 `location` 是否返回了 200。

### 22.3.2 使用 document\_root

示例: 从 `html` 路径载入 `Lua` 代码生成内容。

```
http {
    server {
        location / {
```

```

    root html;
    content_by_lua 'local template = require "resty.template" template.
render("view.html", { message = "Hello, World!" })';
}
}
}

```

### 22.3.3 使用 template\_root

示例：从 /usr/local/openresty/nginx/html/templates 路径载入 Lua 代码。

```

http {
    server {
        set $template_root /usr/local/openresty/nginx/html/templates;
        location / {
            root html;
            content_by_lua ' local template = require "resty.template" template.
render("view.html", { message = "Hello, World!" })';
        }
    }
}

```

### 22.3.4 使用 template\_location

示例：使用 ngx.location.capture 从 /templates 读取内容。

```

http {
    server {
        set $template_location /templates;
        location / {
            root html;
            content_by_lua 'local template = require "resty.template" template.
render("view.html", { message = "Hello, World!" })';
        }
        location /templates {
            internal;
            alias html/templates/;
        }
    }
}

```

## 22.4 Lua API

本节介绍 lua-resty-template 提供的 Lua API。

### 1. boolean template.caching(boolean or nil)

说明：使能或禁用模板缓存，如果没有参数直接调用，则返回当前模板缓存的状态。如果默认模板缓存使能，可能因为开发的原因或低内存占用的原因禁用它。

```
local template = require "resty.template"
```



```
template.caching()-- Disable template caching
template.caching(false)-- Enable template caching
template.caching(true)
```

如果模板已经在编译时被缓存了，将返回缓存版本，也可以通过调用 `template.cache={}` 丢弃原来的缓存并使之重新编译。

## 2. table template.new(view, layout)

说明：使用默认的上下文建立一个新的模板实例。返回一个表，只有一个方法是 `render`。表还有 `_tostring` 元表操作定义。`view` 参数和 `layout` 参数可以是字符串或文件路径，`layout` 可以是之前通过 `template.new` 创建的表。

```
local view = template.new"template.html"
local view = template.new("view.html", "layout.html")
local view = template.new[["<h1>{{message}}</h1>"]]
local view = template.new[["<h1>{{message}}</h1>"]], [["<html><body>  {{*view*}}</body></html>"]])
```

例如：

```
local template = require "resty.template"
local view = template.new"view.html"
view.message = "Hello, World!"
view:render()
view:render{ title = "Testing lua-resty-template" }
view:render(setmetatable({ title = "Testing lua-resty-template" }, { _index = view } ))-- To get rendered template as a string, you can use tostringlocal result = tostring(view)
```

## 3. function, boolean template.compile(view, key, plain)

说明：解析、编译、缓存（如果缓存使能）一个模板，编译成功的模板返回一个函数。函数以上下文为参数，以字符串返回渲染的模板。可以传递 `key` 参数作为缓存 `key`，如果缓存 `key` 没有提供 `view`，则其将只作为一个缓存 `key`。如果缓存 `key` 没有缓存模板，缓存将不会检查，并且返回结果函数，并且不被缓存。可以可选传入 `plain` 的值，对于 `plain` 模式的字符串，可以传入 `true`（将跳过 `template.load`，并且跳过 `template.parse` 阶段的二进制块检测）。

```
local func = template.compile("template.html")
local func = template.compile[["<h1>{{message}}</h1>"]]
```

例如：

```
local template = require "resty.template"
local func = template.compile("view.html")
local world = func{ message = "Hello, World!" }
local universe = func{ message = "Hello, Universe!" }print(world, universe)
```

第二个返回值是布尔型，可以忽略掉，或者用它判断返回的函数是否是缓冲的。

#### 4. template.render(view, context, key, plain)

说明：解析、编译、缓存（如果缓存使能），并且输出 ngx.print 或 print 有效的模板。可以选择传递 key 作为一个缓冲 key。如果 plain 是 true，view 将优先使用 plain 纯净模板。将跳过 template.load，并且将跳过 template.parse 阶段的二进制块检测。

```
template.render("template.html", { message = "Hello, World!" }) -- or
template.render([[<h1>{{message}}</h1>]], { message = "Hello, World!" })
```

例如：

```
local template = require "resty.template"
template.render("view.html", { message = "Hello, World!" })
template.render("view.html", { message = "Hello, Universe!" })
```

#### 5. string template.parse(view, plain)

说明：解析模板文件或字条串，生成解析后的模板字符串。该函数在调试模板的时候很有用，可以解析二进制块。如果 plain 是 true，view 将优先使用 plain 纯净模板。将跳过 template.load，并且将跳过 template.parse 阶段的二进制块检测。

```
local t1 = template.parse("template.html") local t2 = template.
parse([[<h1>{{message}}</h1>]])
```

#### 6. string template.precompile(view, path, strip)

说明：将模板当二进制块预编译，二进制块可以写入一个文件输出（并且可以被 Lua 的 loadfile 直接载入）。path 用于指定二进制块输出的文件和路径。strip 设置为 false 会生成调试信息，默认是 true。

```
local view = [[<h1>{{title}}</h1><ul>{% for _, v in ipairs(context) do %}
    <li>{{v}}</li>{% end %}</ul>]]
local compiled = template.precompile(view)
local file = io.open("precompiled-bin.html", "wb")
file:write(compiled)
file:close()
-- Alternatively you could just write (which does the same thing as above)
template.precompile(view, "precompiled-bin.html")

template.render("precompiled-bin.html", {
    title = "Names",
    "Emma", "James", "Nicholas", "Mary"
})
```

#### 7. template.load

说明：载入模板。在 template.parse 之前调用该函数（要保证 template.parse 的 plain 可选参数是 false（默认值））。默认地，有两个 loader，一个是 Lua 用 loader，另一个是 Nginx/OpenResty 用 loader，用户可以覆盖这些函数。例如，需要从数据库中载入一个模板 loader。

Lua 版本的 template.load：

```

local function load_lua(path)
    -- read_file tries to open file from path, and return its content.
    return read_file(path) or path
end

```

默认的 Nginx/OpenResty 的 template.load:

```

local function load ngx(path)
    local file, location = path, ngx.var.template_location
    if file:sub(1) == "/" then
        file = file:sub(2)
    end

    if location and location ~= "" then
        if location:sub(-1) == "/" then
            location = location:sub(1, -2)
        end

        local res = ngx.location.capture(location .. '/' .. file)
        if res.status == 200 then
            return res.body
        end
    end

    local root = ngx.var.template_root or ngx.var.document_root
    if root:sub(-1) == "/" then
        root = root:sub(1, -2)
    end

    -- read_file tries to open file from path, and return its content.
    return read_file(root .. "/" .. file) or path
end

```

如示例所示，如果以字符串提供模板，lua-resty-template 总是尝试从文件（ngx.location.capture）载入模板。如果用户知道自己使用的是字符串，而不是文件，可以在 template.compile、template.render、template.parse 使用 plain 参数。或者替换 template.load。可以使用字符串载入模板或自定义查找模板，比如从数据库里载入模板，如果从 Redis 载入模板：

```

local template = require "resty.template"
template.load = function(s)
    return s
end

```

## 8. template.print

说明：包含一个为 template.render() 或 template.new("example.html") 使用的函数，render() 用来输出结果。默认使用 ngx.print 或 print。可以覆盖这个域，可以使用自己的输出函数替代。该函数在一些框架下使用比较有用。

```

local template = require "resty.template"
template.print = function(s)

```

```

print(s)
print("<!-- Output by My Function -->")
end

```

## 22.5 模板预编译

lua-resty-template 支持模板预编译，这在需要跳过模板解析时非常有用，如不希望在生产系统上发布纯文本内容。预编译可以保证模板不包含一些东西，不去编译部分内容，尽管模板缓存会带来一些性能上的提升。可以在编译脚本里加上预编译功能。

1) 预编译模板，作为二进制文件输出：

```

local template = require "resty.template"
local compiled = template.precompile("example.html", "example-bin.html")

```

2) 载入预编译模板文件，以上下文参数运行：

```

local template = require "resty.template"
template.render("example-bin.html", { "Jack", "Mary" })

```

## 22.6 模板助手

当 lua-resty-template 没有底层的结构或方法扩展时，仍然有一些可能性可以尝试。

- 在全局字符串增加方法、表（但不推荐使用）。
- 在加入上下文之前处理值。
- 创建全局函数。
- 为本地函数创建模板表或上下文表。
- 在表内使用元方法。

编辑全局类型看起来很方便，但是会污染变量空间。

例如，向模板表或上下文表添加 moses：

```

local _ = require "moses"
local template = require "resty.template"
template._ = _

```

然后就可以在模板内使用 \_，例如：

Lua 代码：

```

local template = require "resty.template"
local html = require "resty.template.html"

template.render([[<ul>{% for _, person in ipairs(context) do %}      {%html.
li(person.name)*}{% end %}</ul><table>{% for _, person in ipairs(context) do %}
<tr data-sort="{((person.name or ""):lower())}">      {%html.td{ id = person.
id }(person.name)*}      </tr>{% end %}</table>]], {

```

```

{ id = 1, name = "Emma"},
{ id = 2, name = "James" },
{ id = 3, name = "Nicholas" },
{ id = 4 }
})

```

输出:

```

<ul>
<li>Emma</li>
<li>James</li>
<li>Nicholas</li>
<li />
</ul>
<table>
<tr data-sort="emma">
<td id="1">Emma</td>
</tr>
<tr data-sort="james">
<td id="2">James</td>
</tr>
<tr data-sort="nicholas">
<td id="3">Nicholas</td>
</tr>
<tr data-sort="">
<td id="4" />
</tr>
</table>

```

## 22.7 用法示例

### 22.7.1 引用模板

可以在模板内使用 `{{template}}` 和 `{{template, context}}` 符号引用其他的模板。第一个符号使用当前上下文作为引用模板的上下文，第二个符号使用一个新的上下文替换，下面是一个引用模板并更换上下文的例子。

Lua 代码:

```

local template = require "resty.template"
template.render("include.html", { users = {
    { name = "Jane", age = 29 },
    { name = "John", age = 25 }
}})

```

include.html:

```

<html>
<body>
<ul>
    {% for _, user in ipairs(users) do %}
        {{user.html, user}}
    {% end %}

```

```

        </ul>
    </body>
</html>

```

```

user.html:
<li>User {{name}} is of age {{age}}</li>

```

输出:

```

<html>
<body>
<ul>
<li>User Jane is of age 29</li>
<li>User John is of age 25</li>
</ul>
</body>
</html>

```

## 22.7.2 Layouts 的 views

Layouts (或主页) 可以用来在其他 view 中包装一个新 view。

Lua 代码:

```

local template = require "resty.template"
local layout   = template.new "layout.html"
layout.title   = "Testing lua-resty-template"
layout.view    = template.compile "view.html" { message = "Hello, World!" }
layout:render()-- Or like this
template.render("layout.html", {
    title = "Testing lua-resty-template",
    view  = template.compile "view.html" { message = "Hello, World!" }
})-- Or maybe you like this style more-- (but please remember that context.view
is overwritten on rendering the layout.html)local view    = template.new("view.
html", "layout.html")
view.title      = "Testing lua-resty-template"
view.message    = "Hello, World!"
view:render()-- Well, maybe like this then?local layout    = template.new "layout.
html"
layout.title    = "Testing lua-resty-template"local view    = template.new("view.
html", layout)
view.message    = "Hello, World!"
view:render()

```

view.html:

```

<h1>{{message}}</h1>

```

```

layout.html:
<!DOCTYPE html>
<html>
    <head>
        <title>{{title}}</title>
    </head>
    <body>

```

```

        {*view*}
    </body>
</html>

```

可选的可以这样操作。

Lua 代码:

```

local view      = template.new("view.html", "layout.html")
view.title      = "Testing lua-resty-template"
view.message    = "Hello, World!"
view:render()

```

```

view.html:
{% layout="section.html" %}
<h1>{{message}}</h1>

```

```

section.html:
<div id="section">
    {*view*}
</div>

```

```

layout.html:
<!DOCTYPE html>
<html>
<head>
<title>{{title}}</title>
</head>
<body>
    {*view*}
</body>
</html>

```

输出:

```

<!DOCTYPE html>
<html>
  <head>
    <title>Testing lua-resty-template</title>
  </head>
  <body>
    <div id="section">
      <h1>Hello, World!</h1>
    </div>
  </body>
</html>

```

### 22.7.3 使用 Blocks

blocks 可以在 layouts 的 view 中不同地方移动, layouts 有 blocks 的占位符。

Lua 代码:

```

local view      = template.new("view.html", "layout.html")
view.title      = "Testing lua-resty-template blocks"

```

```

view.message = "Hello, World!"
view.keywords = { "test", "lua", "template", "blocks" }
view.render()

view.html:
<h1>{{message}}</h1>
{-aside-}
<ul>
    {% for _, keyword in ipairs(keywords) do %}
<li>{{keyword}}</li>
    {% end %}
</ul>
{-aside-}

layout.html:
<!DOCTYPE html>
<html>
    <head>
        <title>{*title*}</title>
    </head>

    <body>
        <article>
            {*view*}
        </article>
        {% if blocks.aside then %}
        <aside>
            {*blocks.aside*}
        </aside>
        {% end %}
    </body>
</html>

```

输出:

```

<!DOCTYPE html>
<html>
    <head>
        <title>Testing lua-resty-template blocks</title>
    </head>
    <body>
        <article>
            <h1>Hello, World!</h1>
        </article>
        <aside>
            <ul>
                <li>test</li>
                <li>lua</li>
                <li>template</li>
                <li>blocks</li>
            </ul>
        </aside>
    </body>
</html>

```



### 22.7.4 继承

假如用户有 base.html、layout1.html、layout2.html 和 page.html，且需要这样一个继承关系：base.html → layout1.html → page.html 或 base.html → layout2.html → page.html（实际上不需要限制为 3 层）。

Lua 代码：

```
local res = require"resty.template".compile("page.html"){}
```

```
base.html:
<html lang='zh'>
<head>
<link href="css/bootstrap.min.css" rel="stylesheet">
    { * blocks.page_css * }
</head>
<body>
    { * blocks.main * } <script src="js/jquery.js"></script><script src="js/
bootstrap.min.js"></script>
    { * blocks.page_js * }
</body>
</html>
```

layout1.html:

```
{% layout = "base.html" %}
{-main-}
<div class="sidebar-1">
    { * blocks.sidebar * }
</div>
<div class="content-1">
    { * blocks.content * }
</div>
{-main-}
```

layout2.html:

```
{% layout = "base.html" %}
{-main-}
<div class="sidebar-2">
    { * blocks.sidebar * }
</div>
<div class="content-2">
    { * blocks.content * }
</div>
<div>I am different from layout1</div>
{-main-}
```

page.html:

```
{% layout = "layout1.html" %}
{-sidebar-}
```

```

    this is sidebar
  {-sidebar-}

  {-content-}
    this is content
  {-content-}

  {-page_css-}
  <link href="css/page.css" rel="stylesheet">
  {-page_css-}

  {-page_js-} <script src="js/page.js"></script>
  {-page_js-}

```

或

page.html:

```

{% layout = "layout2.html" %}
  {-sidebar-}
    this is sidebar
  {-sidebar-}

  {-content-}
    this is content
  {-content-}

  {-page_css-}
  <link href="css/page.css" rel="stylesheet">
  {-page_css-}

  {-page_js-} <script src="js/page.js"></script>
  {-page_js-}

```

## 22.7.5 Macros

使用宏处理参数化的 views, 这是 lua-resty-template 中一个很好的特性。

要使用宏, 要首先定义 Lua 代码:

```

template.render("macro.html", {
    item = "original",
    items = { a = "original-a", b = "original-b" }
})

```

macro-example.html:

```

{% local string_macro = [[<div>{{item}}</div>]] %}
{* template.compile(string_macro)(context) *}
{* template.compile(string_macro){ item = "string-macro-context" } *}

```

输出:

```

<div>original</div>
<div>string-macro-context</div>

```

现在在 macro-example.html 中添加函数宏:

```
{% local function_macro = function(var, el)
    el = el or "div"
    return "<" .. el .. ">{{" .. var .. "}}</" .. el .. ">\n"end %}
```

```
{* template.compile(function_macro("item"))(context) *}
{* template.compile(function_macro("a", "span"))(items) *}

```

输出:

```
<div>original</div>
<span>original-a</span>
```

但是,这个方法比较复杂,下面尝试另外一个函数宏:

```
{% local function function_macro2(var)
    return template.compile("<div>{{" .. var .. "}}</div>\n")end %}
{* function_macro2 "item" (context) *}
{* function_macro2 "b" (items) *}

```

输出:

```
<div>original</div>
<div>original-b</div>
```

这是另外一个:

```
{% function function_macro3(var, ctx)
    return template.compile("<div>{{" .. var .. "}}</div>\n")(ctx or context)end %}
{* function_macro3("item") *}
{* function_macro3("a", items) *}
{* function_macro3("b", items) *}
{* function_macro3("b", { b = "b-from-new-context" }) *}

```

输出:

```
<div>original</div>
<div>original-a</div>
<div>original-b</div>
<div>b-from-new-context</div>
```

宏有点复杂,可以使用 form-renders 或 helpers-macros 实现参数化模板输出。需要注意的是,在代码块中({% and %}))不能有 %,但是可以用级联:“%” .. “}”。

## 22.7.6 调用模板中的方法

可以调用模板中的字符串方法(或其他表方法)。

Lua 代码:

```
local template = require "resty.template"
template.render([[<h1>{{header:upper()}}</h1>]], { header = "hello, world!" })
```

输出:

```
<h1>HELLO, WORLD!</h1>
```

### 22.7.7 模板内嵌的 Angular 或其他标签 / 模板

一些情况下需要混合和匹配其他模板（如果客户端调用 Angular 类的 JS 模板），客户端和服务端的 lua-resty-templates 类模板混合，需要这样的模板：

```
<html ng-app>
  <body ng-controller="MyController">
    <input ng-model="foo" value="bar">
    <button ng-click="changeFoo()">{{buttonText}}</button><script
      src="angular.js">
    </body>
</html>
```

可以看到，为 Angular 模板准备了一个 {{buttonText}}，不是为 lua-resty-template 准备的。可以用 {-verbatim-} 或 {-raw-} 处理代码：

```
{-raw-}
<html ng-app>
<body ng-controller="MyController">
<input ng-model="foo" value="bar">
<button ng-click="changeFoo()">{{buttonText}}</button><script src="angular.js">
</body>
</html>
{-raw-}
```

或 ({{head.html}}) 是 lua-resty-template 处理的)：

```
<html ng-app>
  {{head.html}}
  <body ng-controller="MyController">
    <input ng-model="foo" value="bar">
    <button ng-click="changeFoo()">{-raw-}{{buttonText}}{-raw-}</button><script
      src="angular.js">
    </body>
  </html>
```

也可以使用短的转义字符：

```
...
<button ng-click="changeFoo()">\{{buttonText}}</button>
...
```

### 22.7.8 模板内嵌的 Markdown

如果要在模板内嵌 Markdown（和 SmartyPants），可以使用 lua-resty-hoedown（依赖 LuaJIT）。下面是一个使用例子。

Lua 代码：

```
local template = require "resty.template"
template.markdown = require "resty.hoedown"

template.render[=[<html><body>{*markdown[[#Hello, WorldTesting Markdown.]]*}
  </body></html>]=]
```

输出:

```
<html>
  <body>
    <h1>Hello, World</h1>

    <p>Testing Markdown.</p>
  </body>
</html>
```

也可以添加 lua-resty-hoedown 文档中的配置参数, 还可以使用 SmartyPants。

Lua 代码:

```
local template = require "resty.template"
template.markdown = require "resty.hoedown"

template.render[=[<html><body>{*markdown([[#Hello, WorldTesting Markdown with
    "SmartyPants"...]], { smarty_pants = true })*}</body></html>]=]
```

输出:

```
<html>
  <body>
    <h1>Hello, World</h1>

    <p>Testing Markdown with &ldquo;SmartyPants&rdquo;&hellip;</p>
  </body>
</html>
```

## 22.7.9 LSP

LSP (Lua Server Pages) 跟 PHP、ASP、JSP 类似, 使用 lua-resty-template 实现是比较简单的。跟 ASP 一样, LSP 由后端服务对前端页面进行格式化。

nginx.conf:

```
http {
    init_by_lua 'require "resty.core" template = require "resty.template" template.
    caching(false)';
    server {
        location ~ /\.lsp$ {
            default_type text/html;
            content_by_lua 'template.render(ngx.var.uri)';
        }
    }
}
```

上面配置文件在 Lua 环境里创建了一个全局的 template 变量 (也许这不是期望的), 同样创建了一个 location 用于匹配所有的 LSP 文件, 下面渲染模板。

假设一个 index.lsp 请求。

```
index.lsp
{%
```

```

layout = "layouts/default.lsp"
local title = "Hello, World!"
%}
<h1>{{title}}</h1>

```

可以看到这些文件包含一些小的 view(<h1>{{title}}</h1>), 包含一些我们想要运行的 Lua 代码。如果需要一个前端 layout 纯的代码文件, layout 变量应该已经在 view 中定义了。看下其他的文件。

```

layouts/default.lsp
<html>
  {(include/header.lsp)}
  <body>
    {*view*}
  </body>
</html>

```

这里需要一个层去装饰 index.lsp, 但是已经包含了, 例如:

```

include/header.lsp
<head>
<title>Testing Lua Server Pages</title>
</head>

```

这是静态的数据。

最后的输出应该是这样的:

```

<html>
  <head>
    <title>Testing Lua Server Pages</title>
  </head>
  <body>
    <h1>Hello, World!</h1>
  </body>
</html>

```

lua-resty-template 可以简单使用, 也可以复杂使用。只要在 root 目录保存文件, 就使用通常的保存、刷新的开发风格。服务将文件保存时自动装载新文件并重新载入模板 (如果缓存关闭)。如果希望传递参数到 layouts 或向上下文表添加材料 (看下面的例子):

```

{%
layout = "layouts/default.lsp"
local title = "Hello, World!"
context.title = 'My Application - ' .. title
%}
<h1>{{title}}</h1>

```

## 22.8 FAQ

### (1) 如何清除模板缓存

如果使能了缓存，lua-resty-template 会自动缓存模板的结果，可以通过 `template.cache = {}` 清理缓存。

## (2) lua-resty-template 用户

jd.com – 京东商城

## 22.9 小结

大型网站为了提高并发性，往往使用网页静态化技术，以方便使用反向代理技术，并进行 CDN 加速。小型网站往往使用动态网页技术，但这并不适用于大型网站。lua-resty-template 模板定位的是这样一个中间地带，既提供了动态网页能力，又不需要经常变动。

本章详细介绍了模板类的使用方法，并针对每个特性给出了详细的用法示例。通过本章的学习，可以了解并掌握模板化页面的相关知识。

## WebSocket 的使用

WebSocket protocol 是 HTML5 一种新的协议。它实现了浏览器与服务器全双工通信 (full-duple)，一开始的握手需要借助 HTTP 请求完成。

在 WebSocket 出现之前，网站为了实现即时通信，所用的技术是轮询 (polling)。轮询是在特定的时间间隔 (如每 1 秒) 内，由浏览器对服务器发出 HTTP request，然后由服务器返回最新的数据给客户端的浏览器。这种传统的 HTTP request 的模式带来很明显的缺点：浏览器需要不断地向服务器发出请求，然而 HTTP request 的 header 是非常长的，其中包含的有用数据可能只是一个很小的值，这样会占用很多的带宽。

而比较新的实现轮询效果的技术是 Comet，它使用了 AJAX。这种技术虽然可达到全双工通信，但依然需要发出请求。

在 WebSocket API 中，浏览器和服务器只需要做一个握手的动作，两者之间就会形成了一条快速通道，从而可以直接互相传送数据。WebSocket 协议为我们实现即时服务带来了两大好处。

- Header: 互相沟通的 Header 是很小的，大概只有 2 Bytes。
- Server Pus: 服务器的推送。服务器不再被动地在接收到浏览器的 request 之后才返回数据，而是在有新数据时就主动推送给浏览器。

lua-resty-websocket 库在 ngx\_module 的 cosocket API 上实现了非阻塞的 WebServer 服务端和非阻塞的 WebSocket 客户端。

库在 OpenResty 中默认是使能的，可以在 ./configure 时通过 --without-lua\_resty\_websocket 选项禁用。



## 23.1 示例

示例：lua-resty-websocket 模块的使用方法。

```

local server = require "resty.websocket.server"

local wb, err = server:new{
    timeout = 5000, -- in milliseconds
    max_payload_len = 65535,
}
if not wb then
    ngx.log(ngx.ERR, "failed to new websocket: ", err)
    return ngx.exit(444)
end

local data, typ, err = wb:recv_frame()

if not data then
    ngx.log(ngx.ERR, "failed to receive a frame: ", err)
    return ngx.exit(444)
end

if typ == "close" then
    -- send a close frame back:

    local bytes, err = wb:send_close(1000, "enough, enough!")
    if not bytes then
        ngx.log(ngx.ERR, "failed to send the close frame: ", err)
        return
    end
    local code = err
    ngx.log(ngx.INFO, "closing with status code ", code, " and message ", data)
    return
end

if typ == "ping" then
    -- send a pong frame back:

    local bytes, err = wb:send_pong(data)
    if not bytes then
        ngx.log(ngx.ERR, "failed to send frame: ", err)
        return
    end
end
elseif typ == "pong" then
    -- just discard the incoming pong frame
end
else
    ngx.log(ngx.INFO, "received a frame of type ", typ, " and payload ", data)
end

wb:set_timeout(1000) -- change the network timeout to 1 second

bytes, err = wb:send_text("Hello world")
if not bytes then
    ngx.log(ngx.ERR, "failed to send a text frame: ", err)

```

```

        return ngx.exit(444)
    end

    bytes, err = wb:send_binary("blah blah blah...")
    if not bytes then
        ngx.log(ngx.ERR, "failed to send a binary frame: ", err)
        return ngx.exit(444)
    end

    local bytes, err = wb:send_close(1000, "enough, enough!")
    if not bytes then
        ngx.log(ngx.ERR, "failed to send the close frame: ", err)
        return
    end
end

```

这个示例代码可以在 nginx.conf 内配置一个 http 模块，在 server 中注册一个 location，注册为 content\_by\_lua\_file，则可以使服务器处理 WebSocket 协议。

nginx.conf:

```

#user nobody;
worker_processes 1;

#error_log logs/error.log;
#error_log logs/error.log notice;
#error_log logs/error.log info;

#pid logs/nginx.pid;

events {
    worker_connections 1024;
}

http {
    include mime.types;
    default_type application/octet-stream;

    #log_format main '$remote_addr - $remote_user [$time_local] "$request" '
    #                '$status $body_bytes_sent "$http_referer" '
    #                '"$http_user_agent" "$http_x_forwarded_for"';

    #access_log logs/access.log main;

    sendfile on;
    #tcp_nopush on;

    #keepalive_timeout 0;
    keepalive_timeout 65;

    #gzip on;

    server {
        listen 80;
        server_name localhost;

        #charset koi8-r;

```

```

#access_log logs/host.access.log main;

location / {
    root    html;
    index  index.html index.htm;
}

#error_page 404              /404.html;

# redirect server error pages to the static page /50x.html
#
error_page 500 502 503 504 /50x.html;
location = /50x.html {
    root    html;
}

# proxy the PHP scripts to Apache listening on 127.0.0.1:80
#
#location ~ \.php$ {
#    proxy_pass http://127.0.0.1;
#}

location /websocket{
    set $cmd_test "test";
    content_by_lua_file socketsvr.lua;
}
}

```

socketsvr.lua 保存 WebSocket 演示代码。

## 23.2 安装

推荐直接使用最新的 OpenResty 包，默认使用本库。

如果在自己的 Nginx 中使用，ngx\_lua 已经安装，需要确保 ngx\_lua 是 0.9.0 以上版本，同时需要在配置文件里配置 lua\_package\_path 指令：

```

# nginx.conf
http {
    lua_package_path "/path/to/lua-resty-websocket/lib/?.lua;;";
    ...
}

```

然后，就可以在代码里使用库了：

```

local server = require "resty.websocket.server"

```

## 23.3 resty.websocket.server

resty.websocket.server 是服务端模块，使用下面的方法载入模块：

```
local server = require "resty.websocket.server"
```

server 模块提供 WebSocket 定义的帧处理函数。

### 1. new

语法:

```
wb, err = server:new()
wb, err = server:new(opts)
```

说明: 在服务端执行一个 WebSocket 握手操作, 并返回一个 WebSocket 服务对象。出现任何错误, 则返回 nil 和错误描述字符串。

opts 是可选参数:

- max\_payload\_len: 指定发送和接收 WebSocket 帧负载最大长度。
- send\_masked: 标示是否发送 masked WebSocket 帧, true 表示发送, false 表示不发送, 默认不发送。
- timeout: 网络超时值, 以毫秒为单位, 可以通过 set\_timeout 方法改变。这个参数并不影响 WebSocket 握手 HTTP 应答头发送处理, 同时需要配置 send\_time 配置指令以设置 HTTP 超时值。

### 2. set\_timeout

语法:

```
wb:set_timeout(ms)
```

说明: 设置网络操作超时值, 以毫秒为单位。

### 3. send\_text

语法:

```
bytes, err = wb:send_text(text)
```

说明: 以文本类型发送数据帧, 返回 TCP 层实际发送的数量。出现任何错误, 则返回 nil 和错误描述字符串。

### 4. send\_binary

语法:

```
bytes, err = wb:send_binary(data)
```

说明: 以二进制类型发送数据报文, 返回 TCP 层实际发送的数据数。出现任何错误, 则返回 nil 和错误描述字符串。

### 5. send\_ping

语法:

```
bytes, err = wb:send_ping()
```

```
bytes, err = wb.send_ping(msg)
```

说明：发送一个 ping 帧，也可以以 msg 可选参数发送 ping 帧，返回 TCP 层实际发送的字节数。出现任何错误，则返回 nil 和错误描述字符串。

注意，本方法不等待对端返回应答帧。

## 6. send\_pong

语法：

```
bytes, err = wb.send_pong()
bytes, err = wb.send_pong(msg)
```

说明：发送一个 pong 帧，返回 TCP 层实际发送的字节数。出现任何错误，则返回 nil 和错误描述字符串。

## 7. send\_close

语法：

```
bytes, err = wb.send_close()
bytes, err = wb.send_close(code, msg)
```

说明：发送一个 close 帧，可以携带可选的 code 和消息。出现任何错误，则返回 nil 和错误描述字符串。

注意，本方法不等对端返回。

支持的 code 如下：

- 1000：一个正常的关闭。
- 1001：指示端点离开了，指示浏览器或服务器离开了页面。
- 1002：指示端点因为协议错误被中断了。
- 1003：标示端点因为收到了不接受的类型数据而中断。
- 1004：保留。
- 1005：保留而且不允许使用。
- 1006：保留。
- 1007：指示端点收到了数据，包含的消息是不支持的，所以中断。
- 1008：指端点收到了不合数据类型规则的数据而中断。
- 1009：端点接收到的消息太大了。
- 1010：表示端点在尝试和服务尝试协商扩展，服务没有在握手信息里返回。
- 1011：表示请求中遇到了不期望的异常。
- 1015：保留。

## 8. send\_frame

语法：

```
bytes, err = wb.send_frame(fin, opcode, payload)
```

说明：用 `fin`（布尔值）、`opcode` 和 `payload` 发送一个原始的 WebSocket 帧。出现任何错误，则返回 `nil` 和错误描述字符串。成功则返回 TCP 发送的字节数。

该方法可用于控制最大允许的负载长度，可以在 `new` 时传递 `max_payload_len` 选项实现；控制是否发送 `masked` 帧，通过将 `send_masked` 设置为 `true` 实现，默认不发送。

支持的 `opcode`（占 4bit）如下：

- `%x0`：连续帧。
- `%x1`：文本帧。
- `%x2`：二进制帧。
- `%x3-7`：保留。
- `%x8`：连接关闭。
- `%x9`：ping。
- `%xA`：pong。
- `* %xB-F`：保留。

## 9. `recv_frame`

语法：

```
data, typ, err = wb:recv_frame()
```

说明：从线路上接收一个 WebSocket 帧。出现任何错误，则返回 `nil` 和错误描述字符串。

第 2 个返回值总是帧类型，取值为 `continuation`、`text`、`binary`、`close`、`ping`、`pong`、`nil`（表示未知类型）中的一个，其中 `close` 帧将返回 3 个值，第三个为扩展的状态信息，参见 `send_close` 描述。其他帧类型返回载荷和类型。对于分片帧，`err` 返回“again”。

## 23.4 `resty.websocket.client`

载入客户端模块，代码如下：

```
local client = require "resty.websocket.client"
```

下面是一个简单的示例：

```
local client = require "resty.websocket.client"
local wb, err = client:new()
local uri = "ws://127.0.0.1:" .. ngx.var.server_port .. "/s"
local ok, err = wb:connect(uri)
if not ok then
    ngx.say("failed to connect: " .. err)
    return
end

local data, typ, err = wb:recv_frame()
```

```

if not data then
    ngx.say("failed to receive the frame: ", err)
    return
end

ngx.say("received: ", data, " (" , typ, "): ", err)

local bytes, err = wb:send_text("copy: " .. data)
if not bytes then
    ngx.say("failed to send frame: ", err)
    return
end

local bytes, err = wb:send_close()
if not bytes then
    ngx.say("failed to send frame: ", err)
    return
end

```

client 模块提供 WebSocket 定义的帧处理函数。

#### (1) client:new

语法:

```

wb, err = client:new()
wb, err = client:new(opts)

```

说明: 创建一个 WebSocket 客户端对象。出现任何错误, 则返回 nil 和错误描述字符串。

opts 是可选参数:

- max\_payload\_len: 指定发送和接收 WebSocket 帧负载最大长度。
- send\_masked: 标示是否发送 masked WebSocket 帧, true 表示发送, false 表示不发送, 默认不发送。
- timeout: 网络超时值, 以毫秒为单位, 可以通过 set\_timeout 方法改变。这个参数并不影响 WebSocket 握手 HTTP 应答头发送处理, 同时需要配置 send\_time 配置指令以设置 HTTP 超时值。

#### (2) client:connect

语法:

```

ok, err = wb:connect("ws://<host>:<port>/<path>")
ok, err = wb:connect("wss://<host>:<port>/<path>")
ok, err = wb:connect("ws://<host>:<port>/<path>", options)
ok, err = wb:connect("wss://<host>:<port>/<path>", options)

```

说明: 连接到 WebSocket 服务端口, 并且执行握手操作。

在实际解析主机名并连接到远端前, 方法总是在连接池中查找本方法前次调用的空闲连接。

options 是一个可选的 Lua 表, 用于控制连接:

- protocols: 指定当前会话使用的子协议。options 可以是一个 Lua 表, 存放所有子协议的 Lua 字符串。
  - origin: 指定 origin 请求头。
  - pool: 指定连接池名字, 如果未指定, 连接池名字将使用字符串模板生成 <host>:<port>。
  - ssl\_verify: 指示在 wss://scheme 下是否执行 SSL 证书校验。
- SSL 连接模式 (wss://) 需要 ngx\_lua 0.9.11 或 OpenResty 1.7.4.1。

### (3) client:close

语法:

```
ok, err = wb:close()
```

说明: 关闭当前 WebSocket 连接, 如果没有发送 close 帧, 则自动发送 close 帧。

### (4) client:set\_keepalive

语法:

```
ok, err = wb:set_keepalive(max_idle_timeout, pool_size)
```

说明: 将当前连接立即放入 ngx\_lua 的 cosocket 连接池, 可以指定每一个工作进程连接池中连接最大的空闲时间 (毫秒)。成功则返回 1, 出现任何错误则返回 nil 和一个错误描述字符串。

可以把本方法放到任何放置 close 的地方, 连接并不会被关闭, 但是后续的网络操作会得到 closed 错误。连接被置成了 closed 状态。

### (5) client:set\_timeout

语法:

```
wb:set_timeout(ms)
```

说明: 和 resty.websocket.server 端的 set\_timeout 方法一样。

### (6) client:send\_text

语法:

```
bytes, err = wb:send_text(text)
```

说明: 与 resty.websocket.server 对应方法一样。

### (7) client:send\_binary

语法:

```
bytes, err = wb:send_binary(data)
```

说明: 与 resty.websocket.server 对象的 send\_binary 方法一样。

### (8) client:send\_ping

语法:



```
bytes, err = wb:send_ping()
bytes, err = wb:send_ping(msg)
```

说明：与 `resty.websocket.server` 对象的 `send_ping` 方法一样。

#### (9) `client:send_pong`

语法：

```
bytes, err = wb:send_pong()
bytes, err = wb:send_pong(msg)
```

说明：与 `resty.websocket.server` 的 `send_pong` 方法一样。

#### (10) `client:send_close`

语法：

```
bytes, err = wb:send_close()
bytes, err = wb:send_close(code, msg)
```

说明：与 `resty.websocket.server` 中的 `send_close` 方法一样。

#### (11) `client:send_frame`

语法：

```
bytes, err = wb:send_frame(fin, opcode, payload)
```

说明：与 `resty.websocket.server` 对象的 `send_frame` 方法一样。

#### (12) `client:recv_frame`

语法：

```
data, typ, err = wb:recv_frame()
```

说明：与 `resty.websocket.server` 对象的 `recv_frame` 方法一样。

## 23.5 `resty.websocket.protocol`

使用下面代码载入模块：

```
local protocol = require "resty.websocket.protocol"
```

协议模块提供的方法相对比较简单。

#### 1. `recv_frame`

语法：

```
data, typ, err = protocol.recv_frame(socket, max_payload_len, force_masking)
```

说明：接收一个 WebSocket 帧。

#### 2. `build_frame`

语法：

```
frame = protocol.build_frame(fin, opcode, payload_len, payload, masking)
```

说明：创建一个 WebSocket 帧。

### 3.send\_frame

语法：

```
bytes, err = protocol.send_frame(socket, fin, opcode, payload, max_payload_len,
masking)
```

说明：发送一个 WebSocket 帧。

## 23.6 使用注意事项

WebSocket 库使用中需要注意自动错误日志功能和限制条件。

### 1. 自动错误日志功能

默认情况下，ngx\_lua 模块在网络错误发生时会自动进行日志记录。如果已经自己做了错误处理，可以关闭自动错误日志功能，通过如下配置指令：

```
lua_socket_log_errors off;
```

### 2. 限制条件

- 本库不能在 init\_by\_lua、set\_by\_lua、log\_by\_lua、header\_filter\_by\_lua 中使用，因为 ngx\_lua cosocket API 在这时候无效。
- 模块对象不能放在全局变量里，需要放在 local 变量里，因为变量会被其他例程改写。

## 23.7 小结

随着 HTML5 的推广，WebSocket 得到了大量的使用，WebSocket 可以解决安全传输及服务端主动下推数据的功能。本章介绍了 WebSocket 库的 client 和 server 对象，使用两个对象可以分别开发 WebSocket 服务端和客户端程序，并通过具体示例给出了使用方法演示。

## TCP 私有服务器实例

本章描述一个使用 Nginx 架构实现一个私有的 TCP 服务器的实例，这个服务器用于实现物联网网关类设备使用私有协议接入功能。系统定义了基于 JSON 格式的交互协议，使用 TCP 作为传输层协议。服务器使用 stream 模块实现 TCP 服务的功能，因为整体仍是 Nginx 架构，所以整个系统具备高并发处理能力，接入能力以 W 为单位。

ngx\_lua 模块使用例程机制，使得每一个请求都拥有一个独立的例程。因为网关类设备需要具备双向通信能力，所以与服务器是长连接，服务器会不定时将客户端请求转发到网关设备，也会不定期发送各种通知、消息。每个连接会在 VM 中保持一个例程，一个例程为一个连接处理，而我们编程也只需要考虑单个例程的工作流程即可，不用考虑自行编写服务器要考虑的：异步机制、状态机。通过 ngx\_lua 实现的完全是非阻塞异步的服务器，而编写代码只考虑同步风格。

### 24.1 协议

首先为服务定义一个传输协议，JSON 以字符类型、强大的自描述能力、相较于 XML 的简洁性成为我们首选的传输形式，同时 JSON 在 Lua 中有 cJSON 处理库，且设备端也有对应的处理库，方便对协议报文封装。

下面介绍 DDP 协议概况。

#### 24.1.1 协议总体要求

DDP 协议总体要求如下：

- 1) 协议设计为一应一答的交互模式, 请求报文对应应答报文。
- 2) 若请求方 3 次超时没有收到应答, 则需要断开之前的连接, 重新建立连接。
- 3) 超时时间为  $T \times 500$  毫秒,  $T$  为重试次数。
- 4) 协议第一版本不支持加密。
- 5) 协议体完全采用 JSON 封装。
- 6) 协议为同步模式, 不支持异步模式, 即一应一答。
- 7) 错误码遵循 HTTP 的规范, 200 系列表示成功, 300 系列表示重定向, 400 系列表示客户端错误, 500 系列表示服务端错误。DDP 错误均从 X50 开始。
- 8) 为减轻服务器负担, 若设备累积 60 分钟没有收发数据, 则需要重新登录, 以保证链路有效。

### 24.1.2 包头定义

包头携带控制类信息, 主要用命令字控制程序流转, 内容放在包体中。

#### 1. 请求报文

- 1) name: 协议名字, 恒为 DDP。
- 2) version: 版本, 从 1.0 开始, 逢重要更新才增加版本号。
- 3) session: 会话 ID。初始 session 为 0, 登录成功后, 使用服务器分配 session 替换, 直到退出, session 在此期间一直有效。
- 4) command: 命令, 是协议操作命令字。
- 5) content: 协议数据正文, 具体参数由具体协议定义。
- 6) flow: 方向, req——请求, resp——应答。
- 7) sequence: 序号, 可以在命令字空间内排序, 也可以全局排序。0 为请求, 1 为应答。

示例:

```
{
  "Name": "DDP",
  "Version": "1.0",
  "Session": "10010",
  "Command": "003",
  "flow": "0",
  "sequence": "001",
  "Content": "{
    \"devId\": \"922383\",
    \"beginTime\": \"2016/10/18 14:33:43\",
    \"beginTime\": \"2016/10/18 15:33:44\"
  }"
}
```

#### 2. 应答报文

因为协议设计成同步协议, 所以应答包只包含 3 个域, 不包含包头同步信息。

- 1) code: 返回码。
- 2) message: 错误信息或提示信息, 由应答方填写。
- 3) result: 应答包体, 具体由操作字决定。

示例 1:

```
{
  "code": "200",
  "message": "用户登录成功",
  "result": {
    "session": "323721"
  }
}
```

示例 2:

```
{
  "code": "200",
  "message": "Get alarm successfully!",
  "result": {
    "alarm.list": [
      {devId: "10001", time: "2016-10-18 23:53:33", owner: "lmj"},
      {devId: "10001", time: "2016-10-18 23:53:35", owner: "lmj"},
      {devId: "10001", time: "2016-10-18 23:53:37", owner: "lmj"},
      ...
    ]
  }
}
```

### 24.1.3 协议命令

DDP 协议定义了 9 个协议操作, 用于实现网关类设备管理, 分别为登录、退出、数据上报、查询数据、消息、命令、前端配置、配置获取、前端升级。

以数据上报为例, 报文数据格式如下:

```
{
  "gwId": "2233",
  "Data": [
    {"sensor": "01", "value": "10"},
    {"sensor": "02", "value": "97.5"}
  ]
}
```

返回报文无包体。

查询数据操作报文格式如下:

```
Content: {
  Sensors: ["11112222", "33334444", "55556666"]
}
```

应答报文格式如下:

```
Result: {
```

```

"Value": [
  {id: "11112222", value: "134" },
  {id: "33334444", value: "333" },
  {id: "55556666", value: nil }
]
}

```

## 24.2 DDP 系统架构

DDP 系统架构如图 24-1 所示。

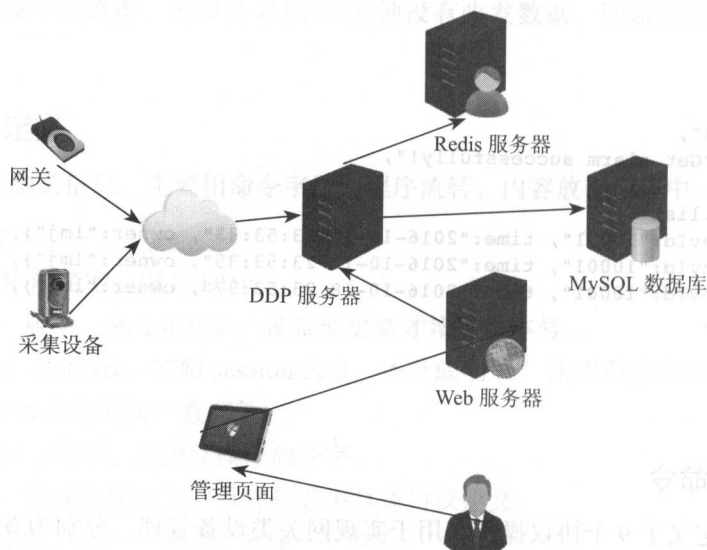


图 24-1 DDP 系统架构

系统整体是一个以 Nginx 为核心的架构。

- DDP 是系统核心，是 DDP 协议处理服务器，也是接入服务器。本系统只演示简单应用，对于集群和负载均衡暂未演示，未来容量不足时可以在前面使用一个 Nginx 服务配置成负载均衡器，管理多台 upstream 型的 DDP 服务器。DDP 服务器基于 Nginx 实现，配合 Redis 服务器实现会话、报警类数据高速缓存。未来 Redis 容量不足时，可以对 Redis 进行主从扩展。
- 系统使用一个 MySQL 存放关系型数据，当 MySQL 容量不足时，可以将其扩展成集群，使用 proxy 实现读写分离。
- Redis 服务器用做系统数据高速缓存服务，缓存会话、最新数据、报警、历史数据等，未来可以进行主从扩展容量。
- Web 服务器用于向用户提供 Web 操作界面，用户可以在管理页面上对网关设备进行操作，所有操作以 Rest 接口形式由 DDP 服务器提供。

## 24.3 DDP 服务实现

DDP 是以 Nginx 为核心实现的, 开发语言为 Lua。

DDP 是基于 stream 模块实现的, 注册了 stream 的 content\_by\_lua\_file 阶段。在 content 处理阶段, 获取 ngx.req.socket 对象, 处理客户端的 socket。根据 Nginx 的工作机制, 每一个连接建立后, 会被创建一个例程来处理这个请求, 对应 ddp.lua 代码中的循环。

在 DDP 的 nginx.conf 中, 还注册了 init\_worker\_by\_lua\_file 阶段, 并注册了一个 init.lua 文件, 用于为每个工作进程初始化一个时钟, 定时清理共享内存, 并定时将缓存中的数据刷新数据库。

### 24.3.1 nginx.conf 配置

nginx.conf 的配置如下:

```
user root;
worker_processes 4;
worker_rlimit_nofile 100000;

error_log logs/error.log;

pid logs/nginx.pid;

events{
    use epoll;
    worker_connections 10000;
}

stream{
    tcp_nodelay on;

    lua_package_path "/usr/local/lib/lua/5.1/?.lua;;";

    server{
        listen 127.0.0.1:1688;
        listen 10.113.141.121:1688;
        listen 120.26.142.207:1688;
        lua_socket_log_errors off;
        content_by_lua_file ddp.lua;
    }
}

http{
    include mime.types;
    default_type text/html;

    access_log off;
    server_tokens off;

    lua_shared_dict gvar 50m;
    lua_shared_dict gmsg 50m;
```

```

lua_shared_dict gkey 200m;
lua_shared_dict gsess 30m;

sendfile      on;
tcp_nopush    on;
tcp_nodelay   on;
open_file_cache max=10240 inactive=60s;
open_file_cache_valid 80s;
open_file_cache_min_uses 1;

keepalive_timeout 0;
chunked_transfer_encoding off;
lua_package_path "/usr/local/lib/lua/5.1/?..lua;;";
init_worker_by_lua_file init.lua;

upstream bk_mysql {
    drizzle_server 10.47.104.71:3306 protocol=mysql dbname=testserver user=ets
                                     password=ManNiu1545~#;
    drizzle_keepalive max=300 overflow=reject mode=single;
}

upstream bk_redis{
    #server 10.175.204.120:6379;
    server 10.47.104.71:6379;
    keepalive 1000;
}

server{
    listen 9000 default so_keepalive=on;
    server_name 10.175.194.47;
    charset utf-8;
    location /redis_set_ex {
        include /usr/local/ip_limit.conf;
        set $key $arg_key;
        set $expire $arg_expire;
        redis2_query set $key $request_body;
        redis2_query expire $key $expire;
        redis2_pass bk_redis;
    }
}
}

```

在 DDP 中，我们使用了 epoll 事件模型，每个工作进程 10 000 个连接，也即连接池中每个进程 1000 个连接：

```

events{
    use epoll;
    worker_connections 10000;
}

```

主机是 4 核的，所以配置了 4 个工作进程：

```
worker_processes 4;
```

下面的 stream 模块定义了一个监听在 1688 端口上的 TCP 服务：



```

stream{
    tcp_nodelay on;

    lua_package_path "/usr/local/lib/lua/5.1/??.lua;;";

    server{
        listen 127.0.0.1:1688;
        listen 10.113.141.121:1688;
        listen 120.26.142.207:1688;
        lua_socket_log_errors off;
        content_by_lua_file ddp.lua;
    }
}

```

24 具体的 DDP 服务代码在 ddp.lua 中。ddp.lua 依据 DDP 协议和定义的业务流程处理核心业务。

下面的 http 模块中定义了 4 个共享内存对象，用于进程间通信以及 Web 端和 DDP 服务间交换命令：

```

http{
    include mime.types;
    default_type text/html;

    access_log off;
    server_tokens off;

    lua_shared_dict gvar 50m;
    lua_shared_dict gmsg 50m;
    lua_shared_dict gkey 200m;
    lua_shared_dict gsess 30m;

    ...
}

```

其他的 HTTP 配置用于为 Web 提供 Rest 接口，配置在 server 下，大概形式如下：

```

location /DDP/upgrade {
    lua_need_request_body on;
    client_max_body_size 50k;
    client_body_buffer_size 50k;
    #access_by_lua_file access.lua;
    content_by_lua '

';
}

location /user/getGateways {
    lua_need_request_body on;
    client_max_body_size 50k;
    client_body_buffer_size 50k;
    #access_by_lua_file access.lua;
    content_by_lua '
    local userId=ngx.var.arg_userId
    ...

```

同时注册了一个 init 事件，在每个工作进程初始化时 init 脚本会被调用，用来注册时钟以管理共享内存：

```
http{
    ...
    init_worker_by_lua_file init.lua;
    ...
}
```

## 24.3.2 init.lua

init.lua 内容：

```
math.randomseed(os.time()+ngx.worker.pid()%100)
local delay = 5+math.floor(math.random()*10000000)%5 -- in seconds
ngx.log(ngx.ERR, "delay=" .. delay)
local handler
handler = function(premature)
    if not premature then
        local gvar = ngx.shared.gvar
        local gmsg = ngx.shared.gmsg
        local gsess = ngx.shared.gsess
        local gkey = ngx.shared.gkey

ngx.log(ngx.ERR, "----timer in-----")
        local cmd='curl -s -m 5 --connect-timeout 2 "http://localhost:9000/
            store_sensor_data?count=1000" >/dev/null'
        local ret = os.execute(cmd)

        --local curl = require "lcurl"
        --local http=curl.easy()
        --http:setopt(curl.OPT_NOSIGNAL,1)
        --http:setopt(curl.OPT_TIMEOUT, 5)
        --http:setopt(curl.OPT_CONNECTTIMEOUT, 2)
        --http:setopt_url('http://127.0.0.1:9000/store_sensor_data?count=1000')
        --http:perform()
        --http:close()
ngx.log(ngx.ERR, "----timer out-----")
        gvar:flush_expired(0)
        gmsg:flush_expired(0)
        gsess:flush_expired(0)
        gkey:flush_expired(0)

        local ok, err = ngx.timer.at(delay, handler)
        if not ok then
            return
        end
    end
end

local ok, err = ngx.timer.at(delay, handler)
if not ok then
    return
end
```

init.lua 在管理进程启动过程中被调用，在管理进程开始 fork 工作进程之前时调用。init 的主要工作有两个：

1) 调用 lcurl 库，定期调用本地的 `http://localhost:9000/store_sensor_data?count=1000` 接口，实现将 Redis 中的传感器数据刷入 MySQL 数据库。因为使用了 Redis 缓存，所以对于传感器实时数据并不实时往数据里刷，而是定时批量写入数据库，这个工作由 init.lua 在定时器中执行。

2) 调用共享内存的 `flush_expired(0)` 方法，清除过期内容。

代码最后，要尝试再次注册时钟函数，使时钟可以持续运行下去。

### 24.3.3 ddp.lua

ddp.lua 是服务器的核心处理代码，演示了简单的 DDP 协议和业务处理流程。代码如下：

```
-----
-- MySQL & Redis function sector
-----
function db_exec(sql)
    local rst = nil
    local rcount = 0
    local data = nil
    local resp = nil
    local i, row, col, val
    local http_lib = require "resty.http"
    local http_conn = http_lib:new()
    http_conn:set_timeout(10000)
    local notify_url = 'http://10.47.101.66:2001/exec_sql?sql=' .. ngx.escape_
        uri(sql)
    local res, err = http_conn:request_uri (notify_url, {method = "GET"})
    resp=res.body
    if resp ~= "" then
        local parser = require "rds.parser"
        res, err = parser.parse(resp)
        if res ~= nil then
            local rows = res.resultset
            if rows ~= nil then
                rcount = #rows
                if rows and rcount > 0 then
                    rst={}
                    for i, row in ipairs(rows) do
                        data={}
                        for col, val in pairs(row) do
                            data[col] = val
                        end
                        rst[i]=data
                    end
                end
            end
        end
    end
    return rst,rcount
end
```

```

end

function redis_set_ex(key,val,expire)
    local http_lib = require "resty.http"
    local http_conn = http_lib:new()
    http_conn:set_timeout(10000)
    local notify_url='http://10.47.101.66:2001/mt_redis_set_ex?key=' .. key ..
        '&expire=' .. expire .. '&val=' .. ngx.escape_uri(val)
    local res, err = http_conn:request_uri (notify_url,{method = "GET"})
end

function redis_expire(key,expire)
    local http_lib = require "resty.http"
    local http_conn = http_lib:new()
    http_conn:set_timeout(10000)
    local notify_url='http://10.47.101.66:2001/redis_expire?key=' .. key ..
        '&expire=' .. expire
    local res, err = http_conn:request_uri (notify_url,{method = "GET"})
end

function redis_persist(key)
    local http_lib = require "resty.http"
    local http_conn = http_lib:new()
    http_conn:set_timeout(10000)
    local notify_url='http://10.47.101.66:2001/redis_persist?key=' .. key
    local res, err = http_conn:request_uri (notify_url,{method = "GET"})
end

function redis_setl(key,val)
    local http_lib = require "resty.http"
    local http_conn = http_lib:new()
    http_conn:set_timeout(10000)
    local notify_url='http://10.47.101.66:2001/mt_redis_setl?key=' .. key
        .. '&val=' .. ngx.escape_uri(val)
    local res, err = http_conn:request_uri (notify_url,{method = "GET"})
end

function redis_set0(key,val)
    local http_lib = require "resty.http"
    local http_conn = http_lib:new()
    http_conn:set_timeout(10000)
    local notify_url='http://10.47.101.66:2001/mt_redis_set0?key=' .. key ..
        '&val=' .. ngx.escape_uri(val)
    local res, err = http_conn:request_uri (notify_url,{method = "GET"})
end

function redis_get(key)
    local resp = nil
    local typ = nil
    local http_lib = require "resty.http"
    local http_conn = http_lib:new()
    http_conn:set_timeout(10000)
    local notify_url='http://10.47.101.66:2001/redis_get?key=' .. key
    local res, err = http_conn:request_uri (notify_url,{method = "GET"})
    resp=res.body
    if resp ~= "" then
        local redis_parser = require("redis.parser")
        res, typ = redis_parser.parse_reply(resp)
    else
        res=nil
    end
end

```

```

    end
    return res,typ
end
function redis_exists(key)
    local resp = nil
    local typ = nil
    local http_lib = require "resty.http"
    local http_conn = http_lib:new()
    http_conn:set_timeout(10000)
    local notify_url='http://10.47.101.66:2001/redis_exists?key=' .. key
    local res, err = http_conn:request_uri (notify_url,{method = "GET"})
    resp=res.body
    if resp ~= "" then
        local redis_parser = require("redis.parser")
        res, typ = redis_parser.parse_reply(resp)
    else
        res=nil
    end
    return res,typ
end
function redis_del(key)
    local http_lib = require "resty.http"
    local http_conn = http_lib:new()
    http_conn:set_timeout(10000)
    local notify_url='http://10.47.101.66:2001/redis_del?key=' .. key
    local res, err = http_conn:request_uri (notify_url,{method = "GET"})
end
function upload_sensor(gwId, sensorId, value)
    local http_lib = require "resty.http"
    local http_conn = http_lib:new()
    http_conn:set_timeout(10000)
    local notify_url='http://10.47.101.66:2001/upload_sensor_data?gwId=' .. gwId
        .. "&sensorId=" .. sensorId .. "&data=" .. value
    local res, err = http_conn:request_uri (notify_url,{method = "GET"})
end
-----
-- Logic sector & Main program           --
-----
local json_data
local cJSON = require("cjson")
cjson.encode_keep_buffer = 0
local gvar = ngx.shared.gvar
local gmsg = ngx.shared.gmsg
--local gsess = ngx.shared.gsess

local stream, err = ngx.req.socket(true)
local data, partial
stream:settimeout(5000)

local vdata={}
local arrayData={}
local sql
local result
local cmd
local session

```

```

local sequence
local content
local flow
local gwId

local bytes
local err

while not ngx.worker.exiting() do
    data, err, partial = stream:receive()

    repeat
        if not data then
            if err ~= 'timeout' then
                ngx.log(ngx.ERR, 'receive error:', err)
                return
            end
        end

        if gwId ~= nil and string.len(gwId)>0 then
            local queryKey="query_request_"..gwId
            local queryRes, queryTyp = redis_get(queryKey)
            if queryRes ~= nil then
                ngx.log(ngx.ERR, "found query_request command.")
                ngx.log(ngx.ERR, queryRes)
                bytes,err=stream:send(queryRes)
                queryRes, queryTyp = redis_del(queryKey)
            end

            local configKey="config_request_"..gwId
            local configRes, configTyp = redis_get(configKey)
            if configRes ~= nil then
                ngx.log(ngx.ERR, "found config_request command.")
                bytes,err=stream:send(configRes)
                configRes, configTyp = redis_del(configKey)
            end

            local getConfigKey="get_config_request_"..gwId
            local getConfigRes, configTyp = redis_get(getConfigKey)
            if getConfigRes ~= nil then
                ngx.log(ngx.ERR, "found get_config_request command.")
                bytes,err=stream:send(getConfigRes)
                getConfigRes, getConfigTyp = redis_del(getConfigKey)
            end

            local commandKey="command_request_"..gwId
            local commandRes, commandTyp = redis_get(commandKey)
            if commandRes ~= nil then
                ngx.log(ngx.ERR, "found command_request command.")
                bytes,err=stream:send(commandRes)
                commandRes, commandTyp = redis_del(commandKey)
            end

            local messageKey="message_request_"..gwId

```

```

local messageRes, messageType = redis_get(messageKey)
if messageRes ~= nil then
    ngx.log(ngx.ERR, "found message_request command.")
    bytes,err=stream:send(messageRes)
    messageRes, messageType = redis_del(messageKey)
end

end

if data == nil or string.len(data)<0 then
    --ngx.log(ngx.ERR, 'data is nil.')
    break
end

ngx.log(ngx.ERR, data)

vdata={}
vdata=cjson.decode(data)
cmd = vdata["command"]
session = vdata["session"]
sequence = vdata["sequence"]
content = vdata["content"]
flow = vdata["flow"]

if cmd == 1 then
    for k,v in pairs(content) do
        gwId = v["gwId"]
    end

    vdata = {}
    vdata["name"] = "DDP"
    vdata["version"] = "1.0"
    vdata["session"] = session
    vdata["sequence"] = sequence
    vdata["command"] = 1
    vdata["flow"] = 1
    result = {}

    sql="SELECT gwName from gateway WHERE sid='\'' .. gwId .. '\''"
    local rs, rcount=db_exec(sql)

    if rs == nil then
        vdata["code"] = 200
        vdata["message"] = "ok"
        result["session"] = "1"
        vdata["result"] = result
        --gssess:set(gwId, 1)
    else
        vdata["code"] = 451
        vdata["message"] = "device not exist!"
        result["session"] = "0"
        vdata["result"] = result
    end

    json_data = cjson.encode(vdata)

```

```

        bytes,err=stream:send(json_data)
    elseif cmd == 2 then
        -- gsess:set(gwid, 0)
        return
    elseif cmd == 3 then
        arrayData={}

        for k,v in pairs(content) do
            gwId = v["gwId"]
            arrayData = v["data"]
        end

        local sensorId
        local sensorValue

        for k,v in pairs(arrayData) do
            sensorId = v["sensor"]
            sensorValue = v["value"]
            -- sql="INSERT INTO sensorData (gwId,sensorId,time,value) VALUES('\''
            .. gwId .. '\'' , '\'' .. sensorId .. '\'' , '\'' .. ngx.localtime()
            .. '\'' , '\'' .. sensorValue .. '\'' )"
            -- local rs, rcount=db_exec(sql)
            --1. store sensor real data to redis for request for client.
            --2. store record key to share memory
            --3. store record to redis
            if sensorId == nil and gwId == nil and sensorValue == nil then
                ngx.log(ngx.ERR, " DDP.lua null")
            else
                --
                ngx.log(ngx.ERR, "DDP.lua gwId=" .. gwId .. " sensorId=" ..
                    sensorId .. " sensorValue=" ..
                    sensorValue)
                local dbres, dberr = upload_sensor(gwId, sensorId,
                    sensorValue)
            end
        end

        end

        vdata = {}
        vdata["name"]="DDP"
        vdata["version"]="1.0"
        vdata["session"]=session
        vdata["sequence"]=sequence
        vdata["flow"]=1
        vdata["code"] = 200
        vdata["message"] = "ok"
        vdata["command"] = 3
        json_data = cjson.encode(vdata)
        bytes,err=stream:send(json_data)
    elseif cmd == 4 then
        if flow == 1 then
            local resultKey="query_response_"..gwId
            redis_set1(resultKey,data)
            ngx.log(ngx.ERR, "query response stored into redis successfully!")
        else
            ngx.log(ngx.ERR, "mistake pdu.")
        end
    end
end

```



```

        end
    elseif cmd == 5 then
        if flow == 1 then
            local resultKey="message_response"..gwId
            redis_set1(resultKey,data)
            ngx.log(ngx.ERR, "message response stored into redis successfully!")
        else
            ngx.log(ngx.ERR, "mistake pdu.")
        end
    elseif cmd==6 then
        if flow == 1 then
            local cmdKey="command_response"..gwId
            redis_set1(cmdKey, data)
            ngx.log(ngx.ERR, "command response stored into redis successfully!")
        else
            ngx.log(ngx.ERR, "mistake pdu.")
        end
    elseif cmd==7 then
        if flow == 1 then
            local cmdKey="config_response"..gwId
            redis_set1(cmdKey, data)
            ngx.log(ngx.ERR, "config response stored into redis successfully!")
        else
            ngx.log(ngx.ERR, "mistake pdu.")
        end
    elseif cmd==8 then
        if flow == 1 then
            local getConfigKey="get_config_response"..gwId
            redis_set1(cmdKey, data)
            ngx.log(ngx.ERR, "get config response stored into redis
successfully!")
        else
            ngx.log(ngx.ERR, "mistake pdu.")
        end
    else
        ngx.log(ngx.ERR, '-----else-----')
    end
end

until true
end

```

### 24.3.4 DDP 代码解析

第一部分是“MySQL & Redis function sector”，这里定义了访问 MySQL、Redis 的共用函数。因为 ddp.lua 是在 stream 模块的 content 阶段运行的，所以不能使用子请求（ngx.location.capture）访问，而使用 resty.http 库访问定义在 2001 端口的数据库 Rest 接口。这里要注意，为了安全性，提供数据访问 Rest 服务的主机应该不配置公网 IP，和 DDP 服务器之间通过局域网通信。Redis 也是相同的道理。这里封装了 Redis 的常用操作，包括 get、set、delete、exist 等。这些函数都是主代码中要使用的。

第二部分是“Logic sector & Main programe”，是 DDP 协议处理的主代码。

```

local json_data
local cJSON = require("cjson")
cJSON.encode_keep_buffer = 0
local gvar = ngx.shared.gvar
local gmsg = ngx.shared.gmsg
--local gsess = ngx.shared.gsess

local stream, err = ngx.req.socket(true)
local data, partial
stream:settimeout(5000)

```

主要部分是通过 `ngx.req.socket(true)` 获取 `stream` 对象。对于收到的客户端请求，Nginx 做好预处理后，在 `content` 处理阶段就会创建本代码的例程，一对请求对应一个例程。这时我们通过 `ngx.req.socket` 得到底层创建好的 TCP 连接，并由此进行后续处理。回想一下我们自己写网络服务器程序的情况：Nginx 前面处理的这些过程，我们都要自己动手编写，基于 `libevent` 的 `epoll` 实现，编写相对复杂的逻辑，最终也是对连接进行收发处理，根据协议进行业务处理。这里，我们通过一行代码就接管了业务的处理，普通的网络及环境处理由 Nginx 框架实现，并全部继承框架的全异步式非阻塞架构、高并发、高可靠性。

```
stream:settimeout(5000)
```

对网络超时值做了设置，然后程序进入主循环：

```

while not ngx.worker.exiting() do
    data, err, partial = stream:receive()

    repeat
        if not data then
            if err ~= 'timeout' then
                ngx.log(ngx.ERR, 'receive error:', err)
                return
            end
        end
    end
    ...
end

```

调用 `stream:receive()` 进行数据接收，实际上 `receive` 是异步的，客户端并不知道数据到达的时间，因为例程的引入，实际上代码运行到这里，如果没有数据由例程被挂起来，直到有数据才被唤醒，继续执行，而工作进程接着处理其他的例程。这样，代码是以同步方式编写的，其他事务是由 `ngx_lua` 底层处理的。

这里的 `repeat` 是用于实现 `break` 的，因为 Lua 里面没有 `break` 语句，所以用 `repeat` 实现 `break` 功能，这样可以在数据检查失败时直接跳出当前循环。

```

if not data then
    if err ~= 'timeout' then
        ...
    end
end

```

这是比较重要的一段代码，Nginx 底层网络出现错误时都会中断连接，唯有 `timeout` 错误不会中断，但是 `timeout` 会经常出现，所以一定要进行检查，发现这种情况就中断当前循

环，处理下一次数据收发。

第三部分，从 Redis 中读取一些命令的代码是用于调试不方便调试的命令的，并不是正式的生产代码。测试时通过浏览器或 CURL 在测试 URL 上传递参数，参数和命令将被写入 Redis，然后在消息循环里检测并返回结果。这时需要在网页的 REST 接口部分代码中使用 TCPSocket API 向 DDP 服务发条命令，激活消息循环，否则就需要等到连接上有数据处理才会被唤醒，有一定延迟。

```
if gwId ~= nil and string.len(gwId)>0 then
    local queryKey="query_request_"..gwId
    local queryRes, queryTyp = redis_get(queryKey)
    if queryRes ~= nil then
        ngx.log(ngx.ERR, "found query_request command.")
        ngx.log(ngx.ERR, queryRes)
        bytes,err=stream:send(queryRes)
        qureyRes, qureyTyp = redis_del(queryKey)
    end

    local configKey="config_request_"..gwId
    local configRes, configTyp = redis_get(configKey)
    if configRes ~= nil then
        ngx.log(ngx.ERR, "found config_request command.")
        bytes,err=stream:send(configRes)
        configRes, configTyp = redis_del(configKey)
    end
end
...

```

第四部分是协议命令字处理部分。这里的代码并没有演示把登录会话保存在共享内存或 Redis 中，下面的例子中将演示这部分应用。

TCP 私有服务器这个实例演示了 CJSON、resty.http、rds.parser、stream 模块、lcurl 库的使用。

### 24.3.5 Redis 和 MySQL 的 location

下面给出 TCP 私有服务器实例中用到的 Redis 和 MySQL 的 location 代码，在 nginx.conf 中实现。

```
user root;
worker_processes 4;
worker_rlimit_nofile 100000;

error_log logs/error.log;

pid logs/nginx.pid;

events{
    use epoll;
    worker_connections 10000;
}
```

```

stream{
    tcp_nodelay on;

    lua_package_path "/usr/local/lib/lua/5.1/??.lua;;";

    server{
        listen 127.0.0.1:1688;
        listen 10.113.141.121:1688;
        listen 120.26.142.207:1688;
        lua_socket_log_errors off;
        content_by_lua_file ddp.lua;
    }
}

http{
    include      mime.types;
    default_type text/html;

    access_log    off;
    server_tokens off;

    lua_shared_dict gvar 50m;
    lua_shared_dict gmsg 50m;
    lua_shared_dict gkey 200m;
    lua_shared_dict gsess 30m;

    sendfile      on;
    tcp_nopush     on;
    tcp_nodelay    on;
    open_file_cache max=10240 inactive=60s;
    open_file_cache_valid 80s;
    open_file_cache_min_uses 1;

    keepalive_timeout 0;
    chunked_transfer_encoding off;
    lua_package_path "/usr/local/lib/lua/5.1/??.lua;;";
    init_worker_by_lua_file init.lua;

    upstream bk_mysql {
        drizzle_server 10.47.120.22:3306 protocol=mysql dbname=testserver user=test
                                password=5~#32Df;
        drizzle_keepalive max=300 overflow=reject mode=single;
    }

    upstream bk_redis{
        server 10.46.102.64:6379;
        keepalive 1000;
    }

    server{
        listen 9000 default so_keepalive=on;
        server_name 10.175.164.33;
        charset utf-8;
        location /redis_set_ex {
            include /usr/local/ip_limit.conf;

```

```

set $key $arg_key;
set $expire $arg_expire;
redis2_query set $key $request_body;
redis2_query expire $key $expire;
redis2_pass bk_redis;
}

location /redis_expire {
    include /usr/local/ip_limit.conf;
    set $key $arg_key;
    set $expire $arg_expire;
    redis2_query expire $key $expire;
    redis2_pass bk_redis;
}

location /redis_persist {
    include /usr/local/ip_limit.conf;
    set $key $arg_key;
    redis2_query persist $key;
    redis2_pass bk_redis;
}

location /redis_set1 {
    include /usr/local/ip_limit.conf;
    set $key $arg_key;
    redis2_query set $key $request_body;
    redis2_query expire $key 86400;
    redis2_pass bk_redis;
}

location /redis_set0 {
    include /usr/local/ip_limit.conf;
    set $key $arg_key;
    redis2_query set $key $request_body;
    redis2_pass bk_redis;
}

location /redis_get {
    include /usr/local/ip_limit.conf;
    set $key $arg_key;
    redis2_query get $key;
    #redis2_query expire $key 86400;
    redis2_pass bk_redis;
}

location /redis_exists {
    include /usr/local/ip_limit.conf;
    set $key $arg_key;
    redis2_query exists $key;
    redis2_pass bk_redis;
}

location /redis_del {
    include /usr/local/ip_limit.conf;
    set $key $arg_key;

```

```

        redis2_query del $key;
        redis2_pass bk_redis;
    }

    location /mt_redis_set_ex {
        include /usr/local/ip_limit.conf;
        #access_by_lua_file access.lua;
        content_by_lua '
            local val=ngx.unescape_uri(ngx.var.arg_val)
            local resp = ngx.location.capture("/redis_set_ex?key=" .. ngx.var.
                arg_key .. "&expire=" .. ngx.var.arg_expire, {
                method = ngx.HTTP_POST, body = val
            })
            ngx.exit(resp.status)
        '
    }

    location /mt_redis_set1 {
        include /usr/local/ip_limit.conf;
        #access_by_lua_file access.lua;
        content_by_lua '
            local val=ngx.unescape_uri(ngx.var.arg_val)
            local resp = ngx.location.capture("/redis_set1?key=" .. ngx.var.
                arg_key, {
                method = ngx.HTTP_POST, body = val
            })
            ngx.exit(resp.status)
        '
    }

    location /mt_redis_set0 {
        include /usr/local/ip_limit.conf;
        #access_by_lua_file access.lua;
        content_by_lua '
            local val=ngx.unescape_uri(ngx.var.arg_val)
            local resp = ngx.location.capture("/redis_set0?key=" .. ngx.var.
                arg_key, {
                method = ngx.HTTP_POST, body = val
            })
            ngx.exit(resp.status)
        '
    }

    location /mysql {
        include /usr/local/ip_limit.conf;
        drizzle_pass bk_mysql;
        drizzle_query $request_body;
        #rds_csv on;
        #rds_csv_field_name_header off;
    }

    location /exec_sql {
        include /usr/local/ip_limit.conf;
        #access_by_lua_file access.lua;
        content_by_lua '

```

```

        local sql=ngx.unescape_uri(ngx.var.arg_sql)
        local resp = ngx.location.capture("/mysql", {
            method = ngx.HTTP_POST, body = sql
        })
        if resp.status ~= ngx.HTTP_OK or not resp.body then
            ngx.exit(resp.status)
        end
        ngx.print(resp.body)
    }
}

```

### 24.3.6 管理页面 REST 操作

管理页面提供了用户操作设备的页面，包括网关添加、删除、修改，传感器添加、删除、修改，命令下发、配置下发、配置获取等。

下面是一个操作的代码示例。

```

location /mdp/getConfig {
    lua_need_request_body on;
    client_max_body_size 50k;
    client_body_buffer_size 50k;
    #access_by_lua_file access.lua;
    content_by_lua '
        local gwId=ngx.var.arg_gwId
        local vdata={}
        local cJSON = require "cjson"

        vdata["name"]="DDP"
        vdata["version"]="v1.0"
        vdata["session"]=1
        vdata["command"]=8
        vdata["flow"]=0
        vdata["sequence"]=1

        vdata["gwId"]=gwId
        vdata["cfgData"]=cfgData

        local jsonRequest=cjson.encode(vdata)

        local devCmdKey="get_config_request_" .. gwId
        local resp = ngx.location.capture("/redis_set1?key=" .. devCmdKey, {method =
            ngx.HTTP_POST, body = jsonRequest})

        local sock = ngx.socket.tcp()
        local ok, err = sock:connect(10.233.122.31, 1688)
        if not ok then
            return
        end
        sock:send("from web.")
        sock:close()
    '
}

```

```

local responseKey="get_config_response_".. gwId
for i = 1, 10, 1 do
    resp = ngx.location.capture("/redis_get?key=" .. responseKey)

    if resp.status == ngx.HTTP_OK and resp.body then
        local parser = require "redis.parser"
        local res, typ = parser.parse_reply(resp.body)

        resp = ngx.location.capture("/redis_del?key=" .. responseKey)
        if res ~= nil then
            ngx.print(res)
            ngx.exit(200)
        end
    end
    break
end
ngx.sleep(1)
end
';
}

```

代码将用户读取配置的请求数据放入 Redis 中，然后创建了一个 TCP 对象向 DDP 服务发送了一条数据，这条数据因为不符合协议格式会被删除，但是会触发一个网络读事件，让相应例程有机会进入工作循环处理请求，如向设备下发命令，当设备处理好后，数据也会被放入对应的 Redis 中，由页面例程取走返回给用户。

后面是检测 Redis 的代码，等待并取出结果。这里也可以使用共享内存实现数据共享。

## 24.4 小结

本章详细介绍了一个基于 stream 模块实现一个私有的 TCP 服务器的示例。这个例子中使用到了 Redis、共享内存、MySQL、RESTFual 等技术，本章给出了这些技术的具体实现代码。

本章介绍了一个基于 JSON 的通信协议，详细介绍了示例中 DDP 服务器的 nginx.conf 文件、init.lua、ddp.lua，并对 DDP 代码进行了解析。通过这个示例，读者可以了解一个完整的典型应用，深入地理解以 Nginx 为核心系统的开发过程。



## WebSocket 接入服务器实战

WebSocket 接入服务器实战这个实例实现了通用的 Web 接口，供客户端调用，同时使用了 WebSocket 协议实现了一个客户长连接接入的机制，客户端和服务端之间可以随时交换各种数据，完全等同于自己使用 C/C++ 等编写的接入业务服务器。

WebSocket 服务器启动后在 9512 端口上监听，接收来自于 WebSocket 客户端的连接。因为 HTML5 支持 WebSocket 协议，所以 Nginx 也支持该协议。客户端和服务端的连接方式是长连接，双方可以在任意时间内主动向对方发起通信。通信协议为 JSON 格式。

这个实例的客户端可以是任意类型，可以接入 APP、桌面程序、智能硬件设备。服务端程序根据协议操作，访问 Redis、MySQL、共享内存等资源和服务。这个服务与自己使用 C++ 开发的业务服务器定位和功能是一样的。借助于 Nginx 的框架，我们不必从头搭建程序框架，不必进行管理接入、连接保持等雷同的处理，而是可以直接实现业务逻辑。

与通常的 Nginx 开发过程一样，首先需要定义 nginx.conf，定义需要监听的 WebSocket 服务。nginx.conf 中定义的系统用到的 location 相当于 RESTful 接口。WebSocket 的处理放在 ws\_svr.lua 中。

### 25.1 nginx.conf 内容

nginx.conf 中配置了全局变量、event 模块、HTTP 模块。使用 epoll 作为 event 模块。HTTP 中定义了供业务层使用的 location，下面将简略给出部分有代表性的一些接口，以演示各种模块的使用方法。其中 ws\_svr 这个 location 用于处理 WebSocket 协议，业务代码在 ws\_svr.lua 中。

关键代码和逻辑在代码中以注释形式给出，由读者参考学习（这是示例代码，并不能直接运行，需要修正和调试后才能运行，这里仅供参考和学习使用）。

nginx.conf 内容：

```

user root;
worker_processes 4;
worker_rlimit_nofile 500000;

error_log logs/error.log;
#error_log logs/error.log notice;
#error_log logs/error.log info;

#pid logs/nginx.pid;

events {
    use epoll;
    worker_connections 50000;
}

http {
    include mime.types;
    default_type text/html;

    #log_format main '$remote_addr - $remote_user [$time_local] "$request" '
    #                '$status $body_bytes_sent "$http_referer" '
    #                '"$http_user_agent" "$http_x_forwarded_for"';

    access_log off;
    server_tokens off;

    sendfile on;
    tcp_nopush on;
    tcp_nodelay on;
    open_file_cache max=10240 inactive=60s;
    open_file_cache_valid 80s;
    open_file_cache_min_uses 1;

    # 这些共享内存供业务部分内部使用
    lua_shared_dict gkey 50m;
    lua_shared_dict gpost 10m;

    keepalive_timeout 0;
    #keepalive_timeout 600s;
    #keepalive_requests 10000;
    request_pool_size 32k;
    chunked_transfer_encoding off;
    #gzip on;
    lua_package_path "/usr/local/lib/lua/5.1/??.lua;;";

    # 下面定义的 MySQL 用在 drizzleMocule2 中的 upstream，即 MySQL 的连接
    upstream bk_mysql {
        drizzle_server 120.33.62.16:3306 protocol=mysql dbname=testserver
                                user=test password=2D45~3#;
        drizzle_keepalive max=300 overflow=reject mode=single;
    }
}

```

```

}
upstream bk_master_db {
    drizzle_server 127.0.0.1:3306 protocol=mysql dbname=testserver user=test
                        password=2D45-3#;
    drizzle_keepalive max=100 overflow=reject mode=single;
}

# 定义的是 Redis 的 upstream, 用于后续访问使用
upstream bk_redis {
    server 120.33.62.16:6379;
    # a pool with at most 1024 connections
    # and do not distinguish the servers:
    keepalive 1000;
}

upstream bk_svr_conf {
    server 120.33.62.18:9511;
    keepalive 1000;
}

server {
    listen          9512 default;
    server_name     120.33.62.16;
    set $pub_ip "120.33.62.16:9512";

    # 系统里还包含了两个转发服务器: 一个使用私有协议, 称为 mts; 另一个使用 HLS 协议, 称为
    # ats
    set $mts "120.33.62.18:8300";
    set $ats "120.33.62.16:8511";
    set $nc "CN";

    #charset koi8-r;
    charset utf-8;
    #chunked_transfer_encoding off;

    #access_log logs/host.access.log main;

    # 定义 Redis 的各种操作
    location /redis_expire {
        include /usr/local/ip_limit.conf;
        set $key $arg_key;
        set $expire $arg_expire;
        redis2_query expire $key $expire;
        redis2_pass bk_redis;
    }

    location /redis_persist {
        include /usr/local/ip_limit.conf;
        set $key $arg_key;
        redis2_query persist $key;
        redis2_pass bk_redis;
    }

    location /redis_set_ex {
        include /usr/local/ip_limit.conf;

```

```

    set $key $arg_key;
    set $expire $arg_expire;
    redis2_query set $key $request_body ex $expire;
    #redis2_query expire $key $expire;
    redis2_pass bk_redis;
}

location /redis_get {
    include /usr/local/ip_limit.conf;
    set $key $arg_key;
    redis2_query get $key;
    #redis2_query expire $key 86400;
    redis2_pass bk_redis;
}

location /redis_exists {
    include /usr/local/ip_limit.conf;
    set $key $arg_key;
    redis2_query exists $key;
    redis2_pass bk_redis;
}

location /redis_del {
    include /usr/local/ip_limit.conf;
    set $key $arg_key;
    redis2_query del $key;
    redis2_pass bk_redis;
}

location /mt_redis_set_ex {
    include /usr/local/ip_limit.conf;
    #access_by_lua_file access.lua;
    content_by_lua '
        local val=ngx.unescape_uri(ngx.var.arg_val)
        local resp = ngx.location.capture("/redis_set_ex?key=" .. ngx.var.
            arg_key .. "&expire=" .. ngx.var.arg_expire, {
            method = ngx.HTTP_POST, body = val
        })
        ngx.exit(resp.status)
    ';
}

location /mt_redis_set0 {
    include /usr/local/ip_limit.conf;
    #access_by_lua_file access.lua;
    content_by_lua '
        local val=ngx.unescape_uri(ngx.var.arg_val)
        local resp = ngx.location.capture("/redis_set0?key=" .. ngx.var.arg_
            key, {
            method = ngx.HTTP_POST, body = val
        })
        ngx.exit(resp.status)
    ';
}

```

```

}

# 定义 MySQL 操作
location /mysql {
    include /usr/local/ip_limit.conf;
    drizzle_pass bk_mysql;
    drizzle_query $request_body;
    #rds_csv on;
    #rds_csv_field_name_header off;
}

```

```

location /exec_sql {
    include /usr/local/ip_limit.conf;
    #access_by_lua_file access.lua;
    content_by_lua '
        local sql=ngx.unescape_uri(ngx.var.arg_sql)
        local resp = ngx.location.capture("/mysql", {
            method = ngx.HTTP_POST, body = sql
        })
        if resp.status ~= ngx.HTTP_OK or not resp.body then
            ngx.exit(resp.status)
        end
        ngx.print(resp.body)
    ';
}

```

## 将请求包体作为 SQL 向主服务器请求。主服务器是 MySQL 集群的写入主服务器，会自动将数据同步到从服务器中，实现各从表数据一致。这个接口的调用者 location 在其他文件中。

```

location /master_db {
    include /usr/local/ip_limit.conf;
    drizzle_pass bk_master_db;
    drizzle_query $request_body;
    #rds_csv on;
    #rds_csv_field_name_header off;
}

```

```

location /mdb_exec {
    include /usr/local/ip_limit.conf;
    #access_by_lua_file access.lua;
    content_by_lua '
        local sql=ngx.unescape_uri(ngx.var.arg_sql)
        local resp = ngx.location.capture("/master_db", {
            method = ngx.HTTP_POST, body = sql
        })
        if resp.status ~= ngx.HTTP_OK or not resp.body then
            ngx.exit(resp.status)
        end
        ngx.print(resp.body)
    ';
}

```

## 这个 location 演示一个与数据库关联的操作，用到了数据库操作、rds.parser、CJSON 操作。这个操作将设备信息写入 dev\_logs 表，未来用于客户端操作统计。

```

location /update_device_info {
    include /usr/local/ip_limit.conf;
}

```

```

lua_need_request_body on;
client_max_body_size 50k;
client_body_buffer_size 50k;
#access_by_lua_file access.lua;
set $debug_log "0";
content_by_lua '
local vdata={}
local json_data=ngx.var.request_body
local cJSON = require "cjson"
vdata=cJSON.decode(json_data)
local sid = vdata["sid"]
local state = vdata["state"]
local geox = vdata["geox"]
local geoy = vdata["geoy"]
local nettype = vdata["nettype"]
local strdata = vdata["strdata"]
local intdata = vdata["intdata"]
local mdltype = vdata["mdltype"]
local rip = vdata["rip"]
if rip == nil then
    rip=""
end
local resp
local sql
if state == 2 then
    sql="INSERT INTO dev_logs(pid, sid, state, logtime, geox, geoy,
        nettype, strdata, intdata, mdltype, rip) VALUES(CRC32(\" ..
        sid .. "\'), \" .. sid .. "\', \" .. state .. \", NOW(), \" ..
        geox .. \", \" .. geoy .. \", \" .. nettype .. \", \" .. strdata ..
        "\', \" .. intdata .. \", \" .. mdltype .. "\', \" .. rip ..
        "\');"
elseif state == 3 then
    sql="INSERT INTO dev_logs(pid, sid, state, logtime, geox, geoy,
        nettype, strdata, intdata, rip) VALUES(CRC32(\" .. sid ..
        "\'), \" .. sid .. "\', \" .. state .. \", NOW(), NULL, NULL,
        NULL, NULL, \" .. rip .. "\');"
elseif state == 4 then
    if ngx.var.debug_log == "1" then
        ngx.log(ngx.ERR, ngx.var.arg_sid .. "-----nettype-----" ..
            nettype .. "-----strdata-----" .. strdata .. "-----intdata-----" .. intdata)
    end
    ngx.exit(ngx.HTTP_OK)
else
    --ngx.say("no acceptable parameter(state)")
    ngx.exit(501)
end
local i
resp = nil
for i = 1, 4, 1 do
    resp = ngx.location.capture("/mysql", {
        method = ngx.HTTP_POST, body = sql
    })
    if resp.status == ngx.HTTP_OK and resp.body then
        break
    end
end

```

```

    ngx.sleep(0.5)
end
if resp.status ~= ngx.HTTP_OK or not resp.body then
    --ngx.say("failed to query mysql")
    ngx.exit(501)
end
local parser = require "rds.parser"
local res, err = parser.parse(resp.body)
if res == nil then
    --ngx.say("failed to parse RDS: " .. err)
    ngx.exit(501)
end

local rows = res.affected_rows
if rows == nil then
    rows=0
end
if rows>0 then
    ngx.say(rows)
    ngx.exit(ngx.HTTP_OK)
else
    --ngx.say("failed to insert")
    ngx.exit(501)
end
';
}

```

# 这是一个报警上报的接口，直接操作 Redis，数据并没有写入数据库，在其他服务的 /update\_alerts 中被批量写入了数据库，参考代码在下节给出。

```

location /update_alert {
    include /usr/local/ip_limit.conf;
    set $sp1 "F";
    set $sp2 "_";
    set_random $idx 0 9;
    set_secure_random alphanum $uuid 27;
    set $redis_key $sp1$sp2$idx$uuid;
    lua_need_request_body on;
    client_max_body_size 50k;
    client_body_buffer_size 50k;
    #access_by_lua_file access.lua;
    content_by_lua "local ev_state=0
    local ev_fid=ngx.var.redis_key
    if ngx.var.arg_state ~= nil then
        ev_state=tonumber(ngx.var.arg_state)
    end
    if ngx.var.arg_key ~= nil then
        if string.len(ngx.var.arg_key) > 0 then
            ev_fid=ngx.var.arg_key
        end
    end
    if string.sub(ev_fid, 1, 2) ~= 'F_' then
        ev_fid = 'F_' .. ev_fid
    end
    local json_data=ngx.var.request_body
    local resp = ngx.location.capture('/redis_set1?key=' .. ev_fid, {

```

```

        method = ngx.HTTP_POST, body = json_data
    })
    ngx.print(ev_fid)
    if ev_state ~= 0 then
        local gsqs = ngx.shared.gsqs
        gsqs:rpunch('fid', ev_fid)
    end
    end
    ";
}

```

# 这是 WebSocket 接口，每个客户端访问本 location 都将会启动一个新例程，合法的用户将建立一个长连接会话。

```

location /ws_svr {
    set $gckey "gctime";
    set $gctime $gckey$pid;
    set $spl "T";
    set $sp2 "_";
    set_secure_random_alphanum $uuuid 28;
    set $session_key $spl$sp2$uuuid;
    lua_socket_log_errors off;
    lua_socket_keepalive_timeout 7200s;
    lua_socket_read_timeout 7200s;

    #access_by_lua_file access.lua;
    content_by_lua_file ws_svr.lua;
    log_by_lua "
    local srcid
    if ngx.var.arg_uid ~= nil then
        srcid=ngx.var.arg_uid
        local gvar = ngx.shared.gvar
        local session_key = gvar:get('user_session_' .. srcid)
        if session_key == ngx.var.session_key then
            gvar:set('user_session_' .. srcid, nil)
            local gex_session = ngx.shared.gex_session
            gex_session:set(srcid, ngx.var.session_key)
        end
    end
    end
    ";
}

```

# 从共享内存中取用用户状态，演示共享内存使用

```

location /user_status {
    include /usr/local/ip_limit.conf;
    default_type 'text/plain';
    lua_need_request_body on;
    client_max_body_size 50k;
    client_body_buffer_size 50k;
    content_by_lua "local srcid
    if ngx.var.arg_uid == nil then
        ngx.exit(ngx.HTTP_INTERNAL_SERVER_ERROR)
    else
        srcid=ngx.var.arg_uid
    end
    local gvar = ngx.shared.gvar
    local user_status = gvar:get('user_status_' .. srcid)

```





```

        data[col] = val
    end
    rst[i]=data
end
end
end
end
end
return rst,rcount
end

# 定义 Redis 操作函数
function redis_set_ex(key,val,expire)
    local http_lib = require "resty.http"
    local http_conn = http_lib:new()
    http_conn:set_timeout(10000)
    local notify_url='http://127.0.0.1:' .. ngx.var.server_port .. '/mt_redis_
        set_ex?key=' .. key .. '&expire=' .. expire .. '&val=' ..
        ngx.escape_uri(val)
    local res, err = http_conn:request_uri (notify_url,{method = "GET"})
end

function redis_get(key)
    local resp = nil
    local typ = nil
    local http_lib = require "resty.http"
    local http_conn = http_lib:new()
    http_conn:set_timeout(10000)
    local notify_url='http://127.0.0.1:' .. ngx.var.server_port .. '/redis_
        get?key=' .. key
    local res, err = http_conn:request_uri (notify_url,{method = "GET"})
    resp=res.body
    if resp ~= "" then
        local redis_parser = require("redis.parser")
        res, typ = redis_parser.parse_reply(resp)
    else
        res=nil
    end
    return res,typ
end

function redis_exists(key)
    local resp = nil
    local typ = nil
    local http_lib = require "resty.http"
    local http_conn = http_lib:new()
    http_conn:set_timeout(10000)
    local notify_url='http://127.0.0.1:' .. ngx.var.server_port .. '/redis_
        exists?key=' .. key
    local res, err = http_conn:request_uri (notify_url,{method = "GET"})
    resp=res.body
    if resp ~= "" then
        local redis_parser = require("redis.parser")
        res, typ = redis_parser.parse_reply(resp)
    else
        res=nil
    end
end

```

```

end
return res, typ
end
function redis_del(key)
    local http_lib = require "resty.http"
    local http_conn = http_lib:new()
    http_conn:set_timeout(10000)
    local notify_url='http://127.0.0.1:' .. ngx.var.server_port .. '/redis_
        del?key=' .. key
    local res, err = http_conn:request_uri (notify_url, {method = "GET"})
end

# 定义使用的局部变量 (这是 Lua 编程尤其要注意的)
local srcid
local user_id
local sub_user={}
local dev_name
local ctrl_access
local func_access
local valid_term
local appid
local openid
local start_ts
local srctype
if ngx.var.arg_uid == nil then
    ngx.exit(ngx.HTTP_INTERNAL_SERVER_ERROR)
else
    srcid=ngx.var.arg_uid
    if string.len(srcid) >= 40 then
        srcid = string.sub(srcid,1,40)
    end
end
local geox=ngx.var.arg_geox
local geoy=ngx.var.arg_geoy
local nettype=ngx.var.arg_nettype
local strdata=ngx.var.arg_chardata
local intdata=ngx.var.arg_intdata
local version=ngx.var.arg_version
local channels=ngx.var.arg_channels
local mdltype=ngx.var.arg_mdltype
local sdkver=ngx.var.arg_sdkver
local vn=ngx.var.arg_vn
if geox == nil then
    geox = 0
elseif geox == '' then
    geox = 0
else
    geox = tonumber(geox)
end
if geoy == nil then
    geoy = 0
elseif geoy == '' then
    geoy = 0
else
    geoy = tonumber(geoy)

```

```

end
if nettype == nil then
    nettype = 0
elseif nettype == '' then
    nettype = 0
else
    nettype = tonumber(nettype)
end
if strdata == nil then
    strdata = ''
elseif string.len(strdata) >= 40 then
    strdata = string.sub(strdata,1,40)
end
if intdata == nil then
    intdata = 0
elseif intdata == '' then
    intdata = 0
else
    intdata = tonumber(intdata)
end
if version == nil then
    version = '1.0'
elseif string.len(version) >= 30 then
    version = string.sub(version,1,30)
end
if version == '' then
    version = '1.0'
elseif string.len(version) >= 30 then
    version = string.sub(version,1,30)
end
if channels == nil then
    channels = 1
elseif channels == '' then
    channels = 1
else
    channels = tonumber(channels)
end
if mdltype == nil then
    mdltype = 'undefined'
end
if sdkver == nil then
    sdkver = ''
end
if vn == nil then
    vn = ''
end

# 载入后面要用到的各种模块
local bit = require("bit")
local json_data
local cJSON = require("cjson")
local rds_parser = require("rds.parser")
local redis_parser = require("redis.parser")
cjson.encode_keep_buffer = 0
local wb,bytes,res,typ,err

```

```

a) local vdata={}
on local resp
a) res, typ = redis_get("S_" .. srcid)
if res ~= nil and typ == redis_parser.BULK_REPLY then
    vdata=cjson.decode(res)
    if vdata["pub_ip"] ~= ngx.var.pub_ip then
        return ngx.redirect('http://'..vdata["pub_ip"].. ngx.var.request
uri,301)
    end
end
res = nil

# 载入 WebSocket 服务模块
local server = require "resty.server"
wb, err = server:new()
if not wb then
    ngx.log(ngx.ERR, 'failed to new websocket: ', err)
    return ngx.exit(444)
end

# 获取几个用到的共享内存
local gpost = ngx.shared.gpost
local gvar = ngx.shared.gvar
local msg_queue = ngx.shared.msg_queue
local gsqs = ngx.shared.gsqs
local gctime=gvar:get(ngx.var.gctime)
if gctime == nil then
    gvar:set(ngx.var.gctime, ngx.now())
end
gctime=gvar:get(ngx.var.gctime)
end

# 这里自己对共享内存做了周期性的管理
if (ngx.now()-gctime)>300 then
    gvar:set(ngx.var.gctime, ngx.now())
    collectgarbage('collect')
    gvar:flush_expired(0)
    msg_queue:flush_expired(0)
    gsqs:flush_expired(0)
end

# 设置 WebSocket 参数
wb:set_timeout(50)
local pri_key=ngx.md5_bin(ngx.header['Sec-WebSocket-Accept'] ..
'e5fb91c5-1b0d-1e1b-9f7a-03e01cblef6e')
--ngx.log(ngx.ERR, 'Sec-WebSocket-Accept: ' .. ngx.header['Sec-WebSocket-
Accept'])
--verify srcid

# 下面是业务操作, 这里并非要展示清楚这个系统是如何工作的, 主要是展示一些技术如何整合和单独使用的
# 方法, 以方便实现自己的系统。
local sql=""
sql="SELECT type,100 as action,sid,'' as userid,'' state,1 as channels,0 as
ctrl_access,appname as dev_name,'' as vn,0 as func_access,0 as valid_term,0
as appid,0 as max_count,0 as max_bps,null(mts_info,'') as mts,null(ats_
info,'') as ats,'' as sub_user,0 as auth_access FROM appserver WHERE
sid='' .. srcid .. '' and (state & 1)=1 union (SELECT a.type,100,a.

```

```

sid,ifnull(a.userid,''),''),,,a.state,ifnull(a.channels,0),ifnull(a.
ctrl_access,1),ifnull(a.devname,''),ifnull(a.vn,''),ifnull(a.func_
access,1),ifnull(unix_timestamp(a.valid_term)-unix_timestamp(),0),ifnull(a.
appid,0),ifnull(a.max_count,20),ifnull(a.max_bps,2048),ifnull(a.mts_
info,''),ifnull(a.ats_info,''),ifnull(b.userid,''),ifnull(b.state,0) FROM
devices a left join dev_viewer b on a.sid=b.devid WHERE a.sid='...' srcid ..
"'" and (a.state & 1)=1) union (SELECT 13,100 as action,sid,'" as userid,'"
.. '"',state,1,0,ifnull(username,''),'','0,0,ifnull(appid,0),0,0,ifnull(mts_
info,''),ifnull(ats_info,''),'','0 FROM users WHERE sid='...' srcid .. '" and
(state & 1)=1)"
end
local i,row,col,val
resp = nil
local rows,rcount
rows,rcount=db_exec(sql)
if rcount == 0 then
return ngx.exit(ngx.status)
end
local session_key = gvar:get('user_session_' .. srcid)
if session_key ~= nil and session_key == ngx.var.session_key then
ngx.log(ngx.ERR, 'user already exists: ' .. srcid)
return ngx.exit(444)
end

vdata = {}
for i, row in ipairs(rows) do
for col, val in pairs(row) do
if i == 1 and col ~= 'sub_user' and col ~= 'auth_access' then
vdata[col] = val
end
end
col=row["sub_user"] .. ""
val=row["auth_access"]
if col ~= "" and bit.band(val, 16) == 16 then
sub_user[col]=val
end
end

if vdata["mts"] == "" then
vdata["mts"] = ngx.var.mts
end
if vdata["ats"] == "" then
vdata["ats"] = ngx.var.ats
end
vdata["stype"] = vdata["type"]
srctype = vdata["type"]
user_id=vdata["userid"]
if user_id ~= '' then
sql="select ifnull(openid,'') as open_id from users where sid='...' user_id ..
"'"
local rs,rcount=db_exec(sql)
if rs ~= nil then
rs=rs[1]
openid=rs["open_id"]
end
end
end

```

```

if openid == nil then
    openid=''
end
dev_name=vdata["dev_name"]
ctrl_access=vdata["ctrl_access"]
func_access=vdata["func_access"]
valid_term=vdata["valid_term"]
appid=vdata["appid"]
vdata["appid"]=nil
start_ts=ngx.time()
vdata["type"] = 2
vdata["skey"] = ngx.var.session_key
vdata["pub_ip"] = ngx.var.pub_ip
vdata["svr_time"] = ngx.utctime()
vdata["pri_key"] = ngx.header['Sec-WebSocket-Accept']
vdata["version"] = version
--if srctype == 1 and vdata["channels"] > 0 then
--    if vdata["vn"] ~= vn then
--        ngx.log(ngx.ERR, srcid .. '-----disconnect,device vn=' .. vn .. ',db
vn=' .. vdata["vn"]])
--        return ngx.exit(444)
--    end
--end
if srctype == 1 then
    if valid_term<10 then
        ngx.log(ngx.ERR, srcid .. '-----disconnect,valid_term=' .. valid_term)
        return ngx.exit(444)
    end
end
vdata["nc"] = ngx.var.nc
if srctype == 1 and vdata["channels"] ~= channels then
    sql="update devices set model='" .. mdltype .. "',nc='" .. ngx.var.nc ..
        "',channels='" .. channels .. " where sid='" .. vdata["sid"] .. "'"
    db_exec(sql)
    mdb_exec(sql)
    vdata["pri_login"] = 1
end
vdata["channels"] = channels
vdata["mdltype"] = mdltype
vdata["sdkver"] = sdkver
gvar:set('user_session_'.. srcid, ngx.var.session_key)
json_data=cjson.encode(vdata)
resp = nil
redis_set_ex("S_" .. srcid,json_data,600)
if srctype == 13 then
    redis_persist("U_" .. srcid)
end
local user_status = gvar:get('user_status_' .. srcid)
if user_status == nil then
    gvar:set('user_status_' .. srcid, 1)
end
local user_count = gvar:get('user_count')
if user_count == nil then
    gvar:set('user_count', 0)
    user_count = 0

```

```

end
local session_exit=1
gvar:incr('user_count',1)
--wb:send_text(srcid .. '|' .. json_data)
vdata = nil
vdata = ngx.encode_te(json_data,pri_key,0)
wb:send_text(srcid .. '|' .. vdata)
vdata = {}
vdata["sid"] = srcid
vdata["state"] = 2
vdata["geox"] = geox
vdata["geoy"] = geoy
vdata["nettype"] = nettype
vdata["strdata"] = strdata
vdata["intdata"] = intdata
vdata["mdltype"] = mdltype
vdata["rip"] = ngx.var.remote_addr
json_data=cjson.encode(vdata)
vdata = nil
local exit_time=0
local next_time=ngx.now() + 120
local update_time=ngx.now()
local data
local packet
local user_array = {}
local http_lib = require "resty.http"
local http_conn = http_lib:new()
http_conn:set_timeout(10000)
resp = nil
http_conn:request_uri ("http://127.0.0.1:" .. ngx.var.server_port .. "/update_
device_info",{method = "POST",body = json_data})
--ngx.log(ngx.ERR, srcid .. '-----connected')
json_data = nil
local def_frame = {}
local partial = nil

# 通过之前的环境初始化工作后，进入主循环，根据网络上的操作进行事件处理循环。
while not ngx.worker.exiting() do
    packet = nil
    data = nil
    data, typ, err, partial = wb:recv_frame(def_frame)
    if not data then
        if wb.fatal then
            if err ~= 'fatal error already happened' and string.find(err, ":
            closed", 1, true) == nil and string.find(err, ": connection
            reset by peer", 1, true) == nil then
                ngx.log(ngx.ERR, srcid .. '-----fatal error(' .. err .. ') on
                rcv_frame')
            end
            break
        else
            if exit_time>0 and ngx.now()>=exit_time then
                ngx.log(ngx.ERR, srcid .. '-----device no response(5 secs) with
                app request')
                break
            end
        end
    end
end

```



```

end
if srctype==1 or srctype==4 then
    if next_time>0 and ngx.now()>=next_time then
        ngx.log(ngx.ERR, srcid .. '-----missing heartbeat packet(120
secs)')
        break
    end
end
--if srctype==2 or srctype==3 or srctype==13 then
--    break
--end
--ngx.log(ngx.ERR, ngx.worker.pid() .. '-----timeout')
def_frame = partial
end
else
    def_frame = {}
    --ngx.log(ngx.ERR, 'receive a frame: ', data)
    if srctype == 1 then
        if ngx.time()>start_ts then
            local secs=ngx.time()-start_ts
            if secs>10 then
                valid_term=valid_term-secs
                start_ts=ngx.time()
            end
        elseif ngx.time()<start_ts then
            start_ts=ngx.time()
        end
        if valid_term<10 then
            ngx.log(ngx.ERR, srcid .. '-----disconnect,valid_term=' ..
valid_term)
            break
        end
    end
    next_time = ngx.now() + 120
    if typ == 'close' then
        ngx.log(ngx.ERR, srcid .. '-----receive websocket closed')
        break
    elseif typ == 'ping' then
        wb:send_pong()
    elseif typ == 'pong' then
    elseif typ == 'text' then
        --wb:send_text(data)
        user_array = {}
        local k
        local key_index
        local find_pos
        local dstid
        local sendtext
        find_pos = string.find(data, '|')
        key_index = 0
        while find_pos ~= nil do
            dstid = string.sub(data,1,find_pos-1)
            if dstid == nil then
                sendtext = data
                break
            end
        end
    end
end

```

```

else
    key_index = key_index + 1
    table.insert(user_array, key_index, dstid)
end
find_pos = find_pos+1
data = string.sub(data,find_pos)
find_pos = string.find(data, '|')
if find_pos == nil then
    sendtext = data
end
end
end
for k,dstid in ipairs(user_array) do
    if dstid ~= nil and string.len(dstid)>0 and dstid ~= srcid then
        if string.len(dstid) >= 5 then
            local dst_exists = gvar:get('user_session' .. dstid)
            if dst_exists ~= nil then
                local offset_w = 'user_msg_' .. dstid
                packet=sendtext
                if string.sub(packet, 1, 1) ~= "{" then
                    packet=ngx.decode_tea(sendtext,pri_key,0)
                end
                msg_queue:rpush(offset_w,srcid .. '|' .. packet)
                packet=nil
            end
        else
            vdata={}
            res, typ = redis_get("S_" .. dstid)
            if res ~= nil and typ == redis_parser.BULK_REPLY then
                vdata=cjson.decode(res)
                if vdata["pub_ip"] ~= ngx.var.pub_ip then
                    packet=sendtext
                    if string.sub(packet, 1, 1) ~= "{" then
                        packet=ngx.decode_tea(sendtext,pri_key,0)
                    end
                    local notify_url='http://' .. vdata["pub_ip"] .. '/send_msg?uid=' .. srcid .. '&did=' .. dstid
                    local notify_body = packet
                    local res, err = http_conn:request_uri(notify_url,{method = "POST",body = notify_body})
                    packet=nil
                end
            end
            wb:send_text(dstid .. '|'.."type":3,"action":902,"result":2}')
        end
        vdata=nil
    end
    wb:send_text(dstid .. '|'.."type":3,"action":902,"result":2}')
    res = nil
end
else
    local notify_url = gpost:get('k' .. dstid)
    if notify_url ~= nil then

```

```

        packet=sendtext
        if string.sub(packet, 1, 1) ~= "{" then
            packet=ngx.decode_tea(sendtext,pri_key,0)
        end
        local notify_body = srcid .. '|' .. packet
        local res, err = http_conn:request_uri (notify_
            url,{method = "POST",body = notify_
                body})
        packet=nil
    end
end
elseif action==102 and cmd_type==2 then
    resp = nil
    local res, err = http_conn:request_uri ("http://
        127.0.0.1:" .. ngx.var.server_port ..
            "/start_live?sid=" .. srcid,{method =
                "POST",body = packet})
elseif action==103 and cmd_type==2 then
    resp = nil
    local res, err = http_conn:request_uri ("http://
        127.0.0.1:" .. ngx.var.server_port .. "/stop_live?sid="
            .. srcid,{method = "POST",body = packet})
elseif action==104 and cmd_type==2 then
    resp = nil
    local res, err = http_conn:request_uri ("http://
        127.0.0.1:" .. ngx.var.server_port .. "/snap_
            picture?sid=" .. srcid,{method = "POST",body = packet})
elseif action==105 and cmd_type==2 then
    resp = nil
    local res, err =http_conn:request_uri ("http://
        127.0.0.1:" .. ngx.var.server_port .. "/"
            upload_video?sid=" .. srcid,{method =
                "POST",body = packet})
elseif action==106 and cmd_type==2 then
    resp = nil
    local res, err =http_conn:request_uri ("http:
        //127.0.0.1:" .. ngx.var.server_port ..
            "/play_video?sid=" .. srcid,{method =
                "POST",body = packet})
elseif action==902 and cmd_type==3 then
    resp = nil
    local res, err =http_conn:request_uri ("http://
        127.0.0.1:" .. ngx.var.server_port
            .. "/update_alert?state=1",{method =
                "POST",body = packet})
elseif action==904 and cmd_type==3 then
    resp = nil
    local res, err =http_conn:request_uri ("http:
        //127.0.0.1:" .. ngx.var.server_port ..
            "/update_p2p_info?sid=" .. srcid,{method
                = "POST",body = packet})
elseif action==905 and cmd_type==3 then
    local func_bit=bit.band(func_access, 16)
    if func_bit == 16 then
        resp = nil
    end
end

```

```

local res, err =http_conn:request_uri ("http://127.0.0.1:" .. ngx.var.server_port .. "/update_vid_info?sid=" .. srcid,{method = "POST",body = packet})
end
--if resp.status == ngx.HTTP_OK then
--    wb:send_text(dstid .. '{"type":3,"action":905,"result":0}')
--else
--    wb:send_text(dstid .. '{"type":3,"action":905,"result":8}')
--end
else
    ngx.log(ngx.ERR, srcid .. '-----unsupported action-----' .. packet)
end
packet=nil
vdata=nil
end
elseif typ == 'binary' then
    --wb:send_binary(data)
end
end
session_key = gvar:get('user_session' .. srcid)
if session_key == nil then
    --session_exit=0
    --wb:send_text(srcid .. '{"type":3,"action":903,"result":0}')
    ngx.log(ngx.ERR, srcid .. '-----missing session key')
    break
elseif session_key~=ngx.var.session_key then
    session_exit=0
    wb:send_text(srcid .. '{"type":3,"action":903,"result":0}')
    ngx.log(ngx.ERR, srcid .. '-----exit old session')
    break
end
local need_send
local v = nil
v = msg_queue:lpop('user_msg' .. srcid)
while v ~= nil do
    need_send = 1
    packet=v
    local find_pos
    local sid
    find_pos = string.find(v, '|')
    if find_pos ~= nil then
        sid = string.sub(v,1,find_pos-1)
        find_pos = find_pos+1
        packet = string.sub(v,find_pos)
        if string.sub(packet, 1, 1) == "{" then
            local cmd_type
            local action
            vdata={}
            vdata=cjson.decode(packet)

```

```

cmd_type = vdata["type"]
action = vdata["action"]
if action==109 and cmd_type==1 then
    exit_time = ngx.now()+5
elseif action==101 and cmd_type==1 then
    if vdata["method"]==0 then
        local channel=vdata["channel"]
        local sql="select func_access from user_config where
            sid='" .. sid .. "'"
        local rs,rcount=db_exec(sql)
        local chan_func=0
        if rs ~= nil then
            rs=rs[1]
            chan_func=rs["func_access"]
        end
        vdata["func_access"]=tonumber(chan_func)
        sql="select net_type,wifi_ip,pass_mask,net_gateway,net_
            dns from devices where sid='" .. srcid .. "'"
        rs,rcount=db_exec(sql)
        if rs ~= nil then
            rs=rs[1]
            if rs["net_type"] ~= nil then
                vdata["net_type"]=rs["net_type"]
            end
            if rs["wifi_ip"] ~= nil then
                vdata["wifi_ip"]=rs["wifi_ip"]
            end
            if rs["pass_mask"] ~= nil then
                vdata["pass_mask"]=rs["pass_mask"]
            end
            if rs["net_gateway"] ~= nil then
                vdata["net_gateway"]=rs["net_gateway"]
            end
            if rs["net_dns"] ~= nil then
                vdata["net_dns"]=rs["net_dns"]
            end
        end
        vdata["channel"]=nil
        if channel == nil then
            sql="select channel,stream,alert_type,pre_secs,rec_
                secs,width,height,overlay_text,overlay_
                pos,fps,bps
                from dev_config where sid='" .. srcid .. "'"
        else
            if string.len(channel) == 0 then
                sql="select channel,stream,alert_type,pre_
                    secs,rec_secs,width,height,overlay_
                    text,overlay_pos,fps,bps
                    from dev_config where sid='" .. srcid .. "'"
            else
                sql="select channel,stream,alert_type,pre_
                    secs,rec_secs,width,height,overlay_
                    text,overlay_pos,fps,bps
                    from dev_config where sid='" .. srcid .. "'"
            end
        end
        and channel in (".. channel ..")

```

```

        end
        rs,rcount=db_exec(sql)
        --vdata["cam_count"]=rcount
        vdata["cam_conf"]=rs
        packet = cJSON.encode(vdata)
    end
elseif action==113 and cmd_type==1 then
    if vdata["on_off"] == 1 then
        func_access = bit.bor(func_access, 4)
        sql="update devices set func_access=" .. func_access ..
            " where sid='" .. vdata["sid"] .. "'"
        db_exec(sql)
    else
        local func_bit=bit.band(func_access, 4)
        if func_bit == 4 then
            func_access = bit.bxor(func_access, 4)
            sql="update devices set func_access=" .. func_access
                .. " where sid='" .. vdata["sid"] .. "'"
            db_exec(sql)
        end
    end
    vdata["func_access"] = func_access
    packet = cJSON.encode(vdata)
elseif action==114 and cmd_type==1 then
    if vdata["on_off"] == 1 then
        func_access = bit.bor(func_access, 16)
        sql="update devices set func_access=" .. func_access ..
            " where sid='" .. vdata["sid"] .. "'"
        db_exec(sql)
    else
        local func_bit=bit.band(func_access, 16)
        if func_bit == 16 then
            func_access = bit.bxor(func_access, 16)
            sql="update devices set func_access=" .. func_access
                .. " where sid='" .. vdata["sid"] .. "'"
            db_exec(sql)
        end
    end
    vdata["func_access"] = func_access
    packet = cJSON.encode(vdata)
elseif action==115 and cmd_type==1 then
    valid_term=vdata["valid_term"]
    start_ts=ngx.time()
elseif action==117 and cmd_type==1 then
    if vdata["on_off"] == 1 then
        func_access = bit.bor(func_access, 1)
        sql="update devices set func_access=" .. func_access ..
            " where sid='" .. vdata["sid"] .. "'"
        db_exec(sql)
    else
        local func_bit=bit.band(func_access, 1)
        if func_bit == 1 then
            func_access = bit.bxor(func_access, 1)
            sql="update devices set func_access=" .. func_access

```

```

        .. " where sid='" .. vdata["sid"] .. "'
        db_exec(sql)
    end
    end
    vdata["func_access"] = func_access
    packet = cJSON.encode(vdata)
end
if need_send == 1 then
    v = packet
    packet = nil
    packet = ngx.encode_tea(v, pri_key, 0)
end
vdata=nil
end
if need_send == 1 then
    packet = sid .. '|' .. packet
end
end
if need_send == 1 then
    bytes, err = wb:send_text(packet)
    if bytes == nil then
        ngx.log(ngx.ERR, srcid .. '-----send error-----')
        v = msg_queue:lpop('user_msg_' .. srcid)
        while v ~= nil do
            v = msg_queue:lpop('user_msg_' .. srcid)
        end
        break
    end
end
packet=nil
v = msg_queue:lpop('user_msg_' .. srcid)
end
local diff_time=ngx.now()-update_time
if diff_time<0 or diff_time>300 then
    redis_expire("S_" .. srcid, 600)
    update_time=ngx.now()
end
end
if session_exit==0 then
    session_key = gvar:get('user_session_' .. srcid)
    if session_key ~= nil then
        gvar:set('user_session_' .. srcid, nil)
    end
    user_status = gvar:get('user_status_' .. srcid)
    if user_status ~= nil then
        gvar:set('user_status_' .. srcid, nil)
    end
    redis_del("S_" .. srcid)
    if srctype == 13 then
        redis_expire("U_" .. srcid, 120)
    end
    vdata = {}
    vdata["sid"] = srcid
    vdata["state"] = 3
    vdata["geox"] = 0

```

```

vdata["geoy"] = 0
vdata["nettype"] = 0
vdata["strdata"] = ""
vdata["intdata"] = 0
vdata["rip"] = ngx.var.remote_addr
json_data=cjson.encode(vdata)
http_conn:request_uri ("http://127.0.0.1:" .. ngx.var.server_port .. "/"
    update_device_info",{method = "POST",body = json_data})
json_data=nil
vdata=nil
end
gvar:incr('user_count',-1)
if session_key == nil or session_exit==0 then
    ngx.sleep(2)
end
wb:send_close()
wb=nil
--ngx.log(ngx.ERR, srcid .. '-----disconnected')

```

## 25.3 update\_alarts 代码

这个 location 供外部时钟性函数调用，用于周期性地将 Redis 中的报警记录写入数据库。可以使用 init.lua 中的时钟，也可以在外部由 cronjob 调用 CURL 工具实现。只是 cronjob 最小时间单位是分钟，一般周期太长了，因为写入的操作不应该超过 1 分钟。

代码主要接收一个 count 参数，决定一次写入多少条记录。下面这个实例还可以进一步优化为使用批量写入语句。

```

location /update_alerts {
    include /usr/local/ip_limit.conf;
    lua_need_request_body on;
    client_max_body_size 50k;
    client_body_buffer_size 50k;
    #access_by_lua_file access.lua;
    content_by_lua 'local key_count
if ngx.var.arg_count == nil then
    ngx.exit(ngx.HTTP_INTERNAL_SERVER_ERROR)
else
    key_count=tonumber(ngx.var.arg_count)
end
local resp = ngx.location.capture("/sqs_get?count=" .. key_count)
if resp.status ~= ngx.HTTP_OK or not resp.body then
    --ngx.say("failed to query sqs")
    ngx.exit(501)
end
local json_data=resp.body
local vkey={}
local cjson = require "cjson"
local parser = require "redis.parser"
vkey=cjson.decode(json_data)
key_count=0

```



```

local col,val
for col, val in pairs(vkey) do
    resp = ngx.location.capture("/redis_get?key=" .. val)
    if resp.status == ngx.HTTP_OK and resp.body then
        local res, typ = parser.parse_reply(resp.body)
        if res ~= nil then
            local ev={}
            ev=cjson.decode(res)
            local sid = ev["sid"]
            local ev_time = ev["ev_time"]
            local ev_type = ev["ev_type"]
            local ev_data = ev["ev_data"]
            local ev_value = ev["ev_value"]
            local ev_channel = ev["ev_channel"]
            local ev_picture = ev["ev_picture"]
            local ev_video = ev["ev_video"]
            local ev_state = ev["ev_state"]
            local ev_vsize = ev["ev_vsize"]
            local ev_fid2 = ev["ev_fid2"]
            local thumb_url = ev["thumb_url"]
            local ev_subtype = ev["ev_subtype"]
            local ev_stop = ev["ev_stop"]
            if ev_vsize == nil then
                ev_vsize = 0
            end
            if ev_fid2 == nil then
                ev_fid2 = ""
            end
            if thumb_url == nil then
                thumb_url = ""
            end
            if ev_subtype == nil then
                ev_subtype = 0
            end
            if ev_stop == nil then
                ev_stop = ""
            end
            end
            --local sql="INSERT INTO alt_logs(pid, sid, logtime, evt_time,
            ev_type, evt_data, evt_value, evt_channel, evt_
            picture, evt_video, evt_state, fid, evt_vsize, fid2,
            thumb_url) VALUES(CRC32(\" .. sid .. "\"), \" ..
            sid .. "\", NOW(), date_add(str_to_date(\" .. ev_
            time .. "\", \"%Y-%m-%d %H:%i:%s\"), interval 8 hour),
            \" .. ev_type .. \", \" .. ev_data .. "\", \" .. ev_
            value .. \", \" .. ev_channel .. \", \" .. ev_picture ..
            \" .. ev_video .. "\", \" .. ev_state .. \", \" ..
            \" .. val .. "\", \" .. ev_vsize .. \", \" .. ev_fid2 ..
            \" .. thumb_url .. "\")"
            local sql="INSERT INTO alt_logs(pid, sid, logtime, evt_time,
            ev_type, evt_data, evt_value, evt_channel, evt_
            picture, evt_video, evt_state, fid, evt_vsize, fid2,
            thumb_url, evt_subtype, evt_stop) VALUES(CRC32(\" ..
            sid .. "\"), \" .. sid .. "\", NOW(), str_to_date(\" ..
            .. ev_time .. "\", \"%Y-%m-%d %H:%i:%s\"), \" .. ev_type
            .. \", \" .. ev_data .. "\", \" .. ev_value .. \", \" ..

```

```

        ev_channel .. ", \" .. ev_picture .. "\" .. ev_
        video .. "\" .. ev_state .. "\" .. val .. "\" ..
        ev_vsize .. ", \" .. ev_fid2 .. "\" .. thumb_url ..
        "\" .. ev_subtype .. ", str_to_date(\" .. ev_stop ..
        "\" .. \"%Y-%m-%d %H:%i:%s\")\""
    resp = ngx.location.capture("/mysql", {
        method = ngx.HTTP_POST, body = sql
    })
    if resp.status == ngx.HTTP_OK and resp.body then
        resp = ngx.location.capture("/redis_del?key=" .. val)
    end
    key_count = key_count + 1
end
end
end
ngx.print(key_count)
';
}

```

## 25.4 小结

本章介绍了一个基于 WebSocket 的 Web 服务，这个服务使用了 Redis、MySQL、RDS、JSON、WebSocket、bit 库等技术，使用了基于共享内存的消息队列，并针对这些内容给出了详细的示例，方便深入地理解这些技术。

限于篇幅的原因，本章所举示例做过大量裁剪，不可以直接使用。

## Nginx 应用简述

Nginx 作为一个高速 Web 服务器，用于典型的 HTTP 环境中。本章重点介绍小、中、大型以 Nginx 为核心的系统进化方式，为初入 Nginx 应用开发的使用者提供一个思路，从开始就全面考虑或知道后面系统会遇到的问题更有利于系统的设计和后续的开发。

以 Nginx 为核心的系统进化方式分为 4 种，下面将具体展开论述。

### 26.1 简单系统

第一种情况是使用 Nginx 做负载均衡或直接在 Nginx 中做业务，将部分请求根据一致性的哈希算法分配到固定的后端主机上，如 Java 的后面，以 Tomcat 为容器。一般通过挂载 1 个或多个 Tomcat 实例实现后端 Java 的大并发。这种系统每次都需要访问 MySQL 数据库，数据库压力大，并且当 Tomcat 数量多了并且请求多了，连接数也会不足（虽然 Tomcat 和 MySQL 间是连接池技术），这个时候就需要对数据库做读写分区了。

单数据库系统如图 26-1 所示。

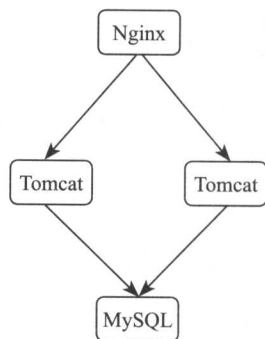


图 26-1 单数据库系统

### 26.2 读写分离系统

数据库主从分区系统如图 26-2 所示。

数据库的写和读分开，组成了簇，相当于 MySQL 集群，有更多的从服务器可以用于读

操作, 服务器之间的数据由 MySQL 自身的主从同步机制实现同步, 但是这有一定的延迟。另外, 对于进一步上升的访问量, 还会遇到速度瓶颈, 这时需要引入缓存技术。

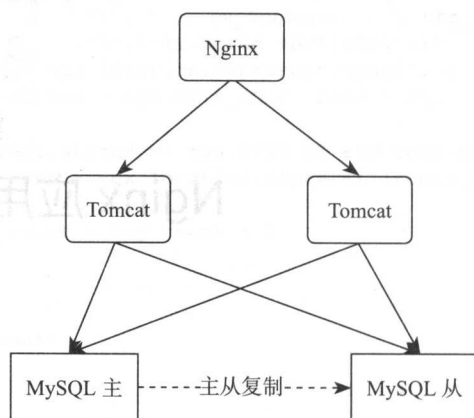


图 26-2 数据库主从分区系统

## 26.3 引入缓存系统

引入了 Redis 缓存的系统结构如图 26-3 所示。

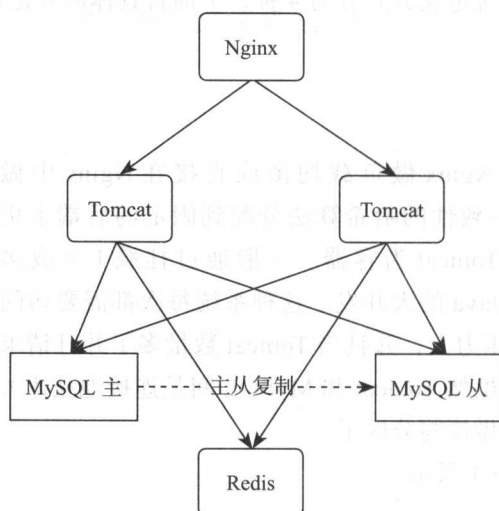


图 26-3 Redis 缓存的系统结构

拥有了缓存则应答前先把数据写入缓存, 再次访问时先访问缓存, 如果命中则不访问数据库, 可以大大减少数据库压力, 因为缓存数据都在内存中, 比从硬盘读取要快, 所以性能要提升。

当数据量大到 Redis 也承受不了时,一般把 Redis 建立成主从系统,类似于 MySQL 主从系统,建立 Redis 集群。

## 26.4 缓存主从系统

引入主从缓存机制的系统如图 26-4 所示。

Redis 主从系统后会有更多的 Redis 实例或服务器为前端提供服务,一般在操作中每个前端对应一个 Redis 实例操作,数据由主从复制机制保证。

当系统进一步扩大以后,Redis 的主从机制也变得复杂起来,因为 Redis 的结构一般是树形的,这时系统的可靠性和维护性就很差,系统需要进一步进化成分区模式。

使用缓存分区的系统结构如图 26-5 所示。

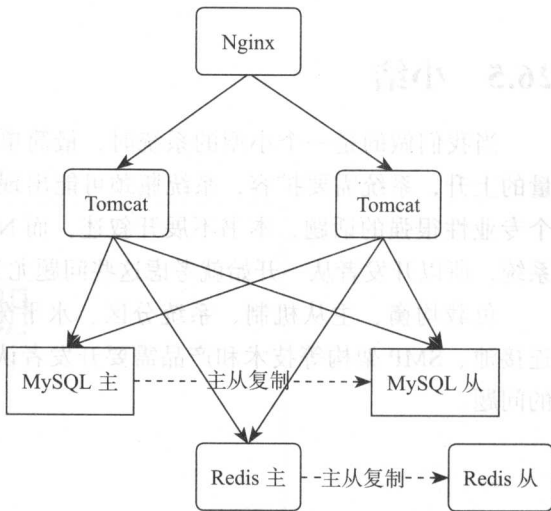


图 26-4 引入主从缓存机制的系统

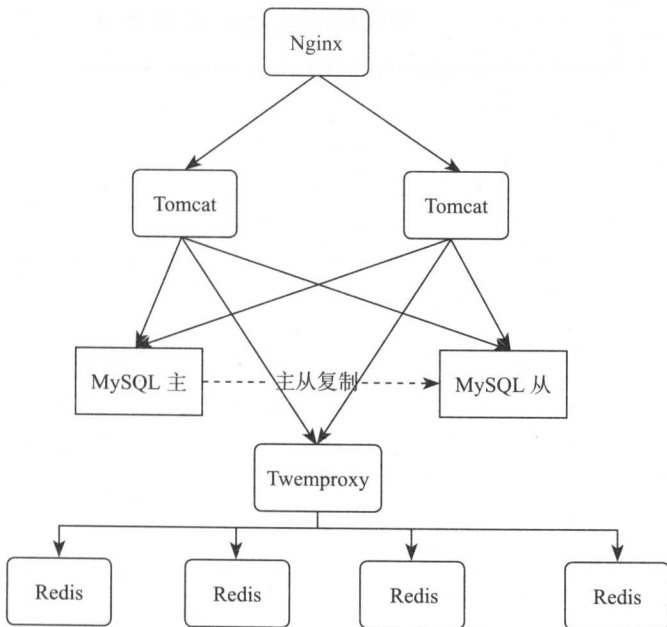


图 26-5 使用缓存分区的系统结构

与主从机制不同,分区系统通过 Twemproxy 中间代理软件进行了分区,实现了集

群。使用一致性的哈希算法，保证每个客户端固定使用 Redis 主机。但这时候中间件主机 Twemproxy 又成为系统的关键节点，需要做备份，同时做一定的负载均衡，还需要使用热备软件，如 LVS/HAProxy。这些内容由读者在项目中自行深入研究。

## 26.5 小结

当我们做的是一个小型的系统时，最简单的架构就可以支撑我们的业务。随着系统容量的上升，系统需要扩容，系统瓶颈可能出现在任意环节。系统扩容以及大系统组建是一个专业性很强的话题，本书不展开叙述，而 Nginx 和 Lua 又主要用于开发高性能、大并发系统，所以开发者从一开始就考虑这些问题尤为重要。

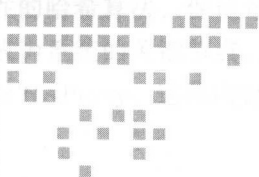
负载均衡、主从机制、系统分区、水平分区、垂直分区、热备、读写分离、多实例、连接池、SMP 架构等技术和产品需要开发者认真考虑，合理地应用这些技术解决系统扩容的问题。



## 第五部分 *Part 5*

# 开发手册

- 第 27 章 ngx\_lua\_module 模块配置指令详解
  - 第 28 章 ngx\_lua API 详解
- .....



## Chapter 27 第 27 章

# ngx\_lua\_module 模块配置指令详解

开 发 手 册

ngx\_lua (ngx\_lua\_module 的简称) 是 Nginx 的内嵌 Lua 模块, 并不在 Nginx 发行的源码中。现在流行的 Nginx 下的 Lua 开发方式是 OpenResty, OpenResty 打包了 Nginx、LuaJIT、ngx\_lua\_module 等, 方便用户使用, 具体请参见第 3 章。

ngx\_lua 由配置指令和 API 两部分组成。本章详细介绍 ngx\_lua 的配置指令。

## 27.1 概述

ngx\_lua 模块通过标准的 Lua 5.1 翻译器或 LuaJIT 2.0/2.1 嵌入 Lua, 在 Nginx 中通过使用子请求, 将 Lua 线程 (Lua 协程) 整合进 Nginx 的事件模型。

与 Apache 的 mod\_lua 及 Lighttpd 的 mod\_magnet 不同, Lua 代码使用整合事件模式可以达到 100% 的非阻塞率, 在网络传输等应用场合用于处理 upstream 的请求, 在如 MySQL、PostgreSQL、Memcached、Redis 或 upstream HTTP 服务中均可高效运作。

至少下列 Lua 库和 Nginx 模块可以被 ngx\_lua 使用:

- lua-resty-memcached;
- lua-resty-mysql;
- lua-resty-redis;
- lua-resty-dns;
- lua-resty-upload;
- lua-resty-websocket;
- lua-resty-lock;



- lua-resty-logger-socket;
- lua-resty-lrucache;
- lua-resty-string;
- ngx\_memc;
- ngx\_postgres;
- ngx\_redis2;
- ngx\_redis;
- ngx\_proxy;
- ngx\_fastcgi。

几乎所有的 Nginx 模块可以被 ngx\_lua 模块通过 ngx.location.capture 或 ngx.location.capture\_multi 使用，但是推荐使用 lua-resty-\* 库代替创建子请求访问 Nginx upstream 模块，因为其更灵活并且更省内存（内部子请求如 location 或命名 location 使用 capture 访问，其他模块使用 lua-resty-\* 库访问，不要混淆了）。

Lua 解释器或 LuaJIT 实例在单 Nginx 工作进程（worker）里被所有的请求共享，但是请求上下文中的 Lua 代码是用轻量级的 Lua 线程（协程）隔离起来运行的。Lua 模块驻留在工作进程的结果就是即使在很重的负载下，Lua 也占用很小的内存。

ngx\_lua 模块放在 Nginx 中 HTTP 子项下，所以只能和 downstream（下游）使用 HTTP 协议簇（HTTP 0.9/1.0/1.1/2.0、WebSocket 等）通信。如果想要使用 TCP 和下游客户端通信，需要使用 ngx\_stream\_lua 模块。

下面的例子演示了一个典型的 Nginx 下的 Lua 代码，这个配置文件修改后可以应用到自己的工程中。这里描述的配置项、Lua API 都是服务于最终的配置脚本或独立的 Lua 文件的。

Nginx 下的 Lua 开发的步骤：

1) 配置 nginx.conf。

2) 注册 Lua 代码。

nginx.conf 内容：

```
# 设置纯 Lua 库的寻找路径 (';;' 是默认的路径):
lua_package_path '/foo/bar/?.lua;/blah/?.lua;';

# 设置 C 语言的 Lua 库路径 (默认路径 ';;'):
lua_package_cpath '/bar/baz/?.so;/blah/blah/?.so;';

server {
    location /lua_content {
        # 默认的 MIME 类型:
        default_type 'text/plain';

        content_by_lua_block {
            ngx.say('Hello,world!')
        }
    }
}
```

```

    }

    location /nginx_var {
        # 默认的 MIME 类型:
        default_type 'text/plain';

        # 尝试使用这个参数访问本 location: /nginx_var?a=hello,world
        content_by_lua_block {
            ngx.say(ngx.var.arg_a)
        }
    }

    location = /request_body {
        client_max_body_size 50k;
        client_body_buffer_size 50k;

        content_by_lua_block {
            ngx.req.read_body() -- 指明去读请求包的包体
            local data = ngx.req.get_body_data()
            if data then
                ngx.say("body data:")
                ngx.print(data)
                return
            end

            -- 包体可以被存放在一个临时文件中:
            local file = ngx.req.get_body_file()
            if file then
                ngx.say("body is in file ", file)
            else
                ngx.say("no body found")
            end
        }
    }

    # Lua 中通过子请求使用非阻塞 I/O
    # (最好的方法是使用 cosocket)
    location = /lua {
        # 默认的 MIME 类型:
        default_type 'text/plain';

        content_by_lua_block {
            local res = ngx.location.capture("/some_other_location")
            if res then
                ngx.say("status: ", res.status)
                ngx.say("body:")
                ngx.print(res.body)
            end
        }
    }

    location = /foo {
        rewrite_by_lua_block {

```

```

        res = ngx.location.capture("/memc",
            { args = { cmd = "incr", key = ngx.var.uri } }
        )
    }

    proxy_pass http://blah.blah.com;
}

location = /mixed {
    rewrite_by_lua_file /path/to/rewrite.lua;
    access_by_lua_file /path/to/access.lua;
    content_by_lua_file /path/to/content.lua;
}

# 在代码路径上使用 Nginx var 变量
# 警告：Nginx var 变量的内容，必须仔细过滤，否则，将会引起巨大的安全风险
location ~ ^/app/([_a-zA-Z0-9/]+) {
    set $path $1;
    content_by_lua_file /path/to/lua/app/root/$path.lua;
}

location / {
    client_max_body_size 100k;
    client_body_buffer_size 100k;

    access_by_lua_block {
        -- 检查客户 IP 地址是否在黑名单中
        if ngx.var.remote_addr == "132.5.72.3" then
            ngx.exit(ngx.HTTP_FORBIDDEN)
        end

        -- 检查 URI 中是否包含敏感词
        if ngx.var.uri and
            string.match(ngx.var.request_body, "evil")
        then
            return ngx.redirect("/terms_of_use.html")
        end

        -- tests passed
    }

    # proxy_pass/fastcgi_pass/etc settings
}

```

## 1. 典型应用

ngx\_lua 可以应用在很多地方，例如：

- 1) 糅合并处理多种 Nginx 上游 (upstream) 输出 (如 Proxy、Drizzle、Postgres、Redis、Memcached 等)。
- 2) 在请求到达 upstream 上游前，做任意复杂的访问控制和安全检查。

3) 任意改变和操作应答头。

4) 从扩展的存储后端匹配信息 (如 Redis、Memcached、MySQL、PostgreSQL), 并且使用该信息选择传输过程中使用的 upstream 后端。

5) 内容使用是同步的, 但仍想在非阻塞访问数据库后端或其他存储的情况下编写任意复杂的 Web 应用。

6) 在 Lua 重写阶段做复杂的 URL 分配。

7) 使用 Lua 实现先进的缓冲机制。

另外, ngx\_lua 允许携带其他多种元素和 Nginx 一起协同工作, 模块提供多种灵活的本, 提供 C 语言级别的性能, 提供 CPU 时间和内存使用的双重高性能, 特别是 LuaJIT 2.X 使能的情况下, 这是其他脚本语言不容易达到的性能。

## 2. 兼容性

最新版本的 ngx\_lua\_module 兼容下列 Nginx 版本:

- 1.11.x (最后测试 1.11.2);
- 1.10.x;
- 1.9.x (最后测试 1.9.15);
- 1.8.x;
- 1.7.x (最后测试 1.7.10);
- 1.6.x。

## 3. 安装

推荐使用 OpenResty, 其打包了 Nginx、ngx\_lua、LuaJIT2.0/2.1 (或可选标准的 Lua 5.1 解释器), 基本安装命令如下:

```
./configure --with-LuaJIT && make && make install
```

有经验的用户可以选择手工安装方式, 将 ngx\_lua 手工编译进 Nginx。

- 安装 LuaJIT2.0 或 2.1 (推荐), 或者 Lua5.1 (目前不支持 Lua5.2)。LuaJIT 可以从 LuaJIT 项目网站上下载 (<http://LuaJIT.org/download.html>)。Lua5.1 从 Lua 网站上下载 (<http://www.lua.org/>)。已经有很多发行版本可用。
- 下载最近版本的 ngx\_devel\_kit (NDK) 模块 ([https://github.com/simpl/ngx\\_devel\\_kit/tags](https://github.com/simpl/ngx_devel_kit/tags))。
- 下载最新版本的 ngx\_lua (<https://github.com/openresty/lua-nginx-module/tags>)。
- 下载最新版本的 Nginx (<http://nginx.org/>)。

编译源码并生成模块:

```
wget 'http://nginx.org/download/nginx-1.11.2.tar.gz'
tar -xzf nginx-1.11.2.tar.gz
cd nginx-1.11.2/
```

编辑系统环境变量：

```
# 设置用于 Nginx 编译环境定位 LuaJIT 2.0 的路径
export LuaJIT_LIB=/path/to/LuaJIT/lib
export LuaJIT_INC=/path/to/LuaJIT/include/LuaJIT-2.0

# 设置用于 Nginx 编译环境定位 LuaJIT 2.1 的路径
export LuaJIT_LIB=/path/to/LuaJIT/lib
export LuaJIT_INC=/path/to/LuaJIT/include/LuaJIT-2.1

# 如果使用 Lua, 配置 Lua 路径
#export LUA_LIB=/path/to/luabin/lib
#export LUA_INC=/path/to/luabin/include

# 下面路径会让 Nginx 安装到 /opt/nginx/
./configure --prefix=/opt/nginx \
    --with-ld-opt="-Wl,-rpath,/path/to/LuaJIT-or-lua/lib" \
    --add-module=/path/to/ngx_devel_kit \
    --add-module=/path/to/lua-nginx-module

make -j2
make install
```

#### 4. 编译动态模块

从 Nginx 1.9.11 开始, 可以把 ngx\_lua 编译成一个动态模块。在 ./configure 命令时使用 --add-dynamic-module=PATH 代替 --add-module=PATH。可以在 nginx.conf 中使用 load\_module 项加载模块。

```
load_module /path/to/modules/ndk_http_module.so; # 假设 NDK 也作为动态模块编译
load_module /path/to/modules/ngx_http_lua_module.so;
```

#### 5. C 宏定义配置

当使用 OpenResty 或者 Nginx 核心编译 ngx\_lua 模块时, 可以定义下列编译器选项:

- NGX\_LUA\_USE\_ASSERT: 定义后, 可以打开 ngx\_lua 模块 C 代码部分断言, 推荐在测试版本或调试代码时使用。打开它将增加运行期负载。
- NGX\_LUA\_ABORT\_AT\_PANIC: 当 Lua/LuaJIT 虚拟机崩溃的时候, ngx\_lua 将通知工作进程优雅地退出。这个宏会立即中断当前的 Nginx 工作进程 (通常会生成一个 core dump 文件)。此宏通常用来调试 Lua 虚拟机崩溃。
- NGX\_LUA\_NO\_EFI\_API: 排除纯 C API, 只使用 Nginx EFI 基础 Lua API (如 lua-resty-core 模块需要此模式)。使能此宏可以使二进制代码更小。

要使能一个或多个宏, 需要在 ./configure 脚本使能扩展的 C 编译器, 例如:

```
./configure --with-cc-opt="-DNGX_LUA_USE_ASSERT -DNGX_LUA_ABORT_AT_PANIC"
```

#### 6. Lua/LuaJIT 字节代码支持

从 v0.5.0rc32 版本开始, 所有的 \*\_by\_lua\_file 配置项 (如 content\_by\_lua\_file) 均支持

装载 Lua5.1 和 LuaJIT2.0/2.1 原始字节码文件。

需要注意的是，LuaJIT 2.0/2.1 的字节码和 Lua5.1 的字节码互不兼容。

如果使用 LuaJIT2.0/2.1，LuaJIT 兼容的字节码必须这样生成：

```
/path/to/LuaJIT/bin/LuaJIT -b /path/to/input_file.lua /path/to/output_file.luac
```

可以加入 -bg 选项以加入调试信息：

```
/path/to/LuaJIT/bin/LuaJIT -bg /path/to/input_file.lua /path/to/output_file.luac
```

在 LuaJIT 官方文档可查阅 -b 选项具体细节：

[http://LuaJIT.org/running.html#opt\\_b](http://LuaJIT.org/running.html#opt_b)

同样地，LuaJIT2.1 和 2.0 的字节码也互不兼容，如果使用 Lua5.1 解释器，字节码需要使用 luac 产生：

```
luac -o /path/to/output_file.luac /path/to/input_file.lua
```

跟 LuaJIT 不同，调试信息默认包含在 Lua5.1 中，可以使用 -s 选项跳过。

```
luac -s -o /path/to/output_file.luac /path/to/input_file.lua
```

企图在 Lua5.1 装入 LuaJIT2.0/2.1 的字节码，将得到下面的类似错误，错误记录在 Nginx 的 error.log 文件中：

```
[error] 13909#0: *1 failed to load Lua inlined code: bad byte-code header in /path/to/test_file.luac
```

通过规范的操作（如 require 和 dofile）装入字节码文件（正确的文件格式），字节码将会按预期工作。

## 7. 环境变量支持

如果需要访问系统环境变量，如 foo，可以使用标准 Lua API 实现：使用 os.getenv。但也必须首先在 nginx.conf 中使用 env 配置项列出此变量，例如：

```
env foo;
```

## 8. HTTP 1.0 支持

HTTP 1.0 协议不支持块输出，并且当应答包体非空的时候，需要一个明确的 Content-Length 头用以支持 HTTP 1.0 的保活。所以，当一个 HTTP 1.0 请求生成时，lua\_http10\_buffering 配置项打开，ngx\_lua 将缓冲 ngx.say 和 ngx.print 的输出，延缓发送输出的包头，直到收到所有的应答包体。ngx\_lua 可以计算包体的总长度，构造一个正式的 Content-Length 头域返回给 HTTP 1.0 客户端。如果在运行的 Lua 代码中直接设置了响应的 Content-Length 头域，尽管 lua\_http10\_buffering 配置项打开，缓冲也会被禁用。

在大的流式应答情况下，禁用 lua\_http\_buffering 配置项是非常重要的，可以降低内存使用率。

注意，普通的 HTTP 测试工具（如 ab 和 http\_load）发出默认的 HTTP 1.0 请求。通过 -0 选项可强制 curl 发出 HTTP 1.0 请求。

### 9. 静态连接纯 Lua 模块

当使用 LuaJIT 2.x 的时候，可能会静态连接纯字节码的 Lua 模块到 Nginx 运行环境。可使用 LuaJIT 将 .lua 模块文件编译成 .o 目标文件，目标文件包含导出的字节码，然后直接将 .o 连接到 Nginx 工程。

下面通过一个示例来论证这项技术。假设我们有一个名为 foo.lua 的 .lua 文件。

```
-- foo.lua
local _M = {}
function _M.go()
    print("Hello from foo")
end

return _M
```

把 .lua 文件编译成 foo.o 文件：

```
/path/to/LuaJIT/bin/LuaJIT -bg foo.lua foo.o
```

.lua 文件告诉 LuaJIT 哪个文件将来会被用到 lua 侧，foo.o 的扩展名 .o 告诉 LuaJIT 使用字节码格式转换 .lua 文件。如果想在结果的字节码中去掉调试信息，使用 -b 选项代替 -bg 选项。

使用字节码模块需要在编译 Nginx 或 OpenResty 的时候，在 ./configure 脚本传入 --with-lid-opt=" foo.o" 选项：

```
./configure --with-lid-opt="/path/to/foo.o" ...
```

完成编译后，可以在 ngx\_lua 的 Lua 代码里这样使用：

```
local foo = require "foo"
foo.go()
```

可以看出，代码不再依赖于 foo.lua 文件，因为它早已经被编译进 Nginx 程序内部。如果想在 require 配置项中使用 . 符号，可以这样使用：

```
local foo = require "resty.foo"
```

这就需要把 foo.lua 文件重命名为 resty\_foo.lua，然后编译成 .o 文件。

在编译 .lua 到 .o 文件的时候用到的 LuaJIT 要和编译 Nginx+ngx\_lua 时用的 LuaJIT 是相同版本。因为若 LuaJIT 不同版本之间的字节码不兼容，则会在 Lua 运行期发生错误，并提示找不到 Lua 模块。

当有多个 .lua 文件要编译和连接的时候，需要把文件名依次放到 --with-lid-opt 选项后面，例如：

```
./configure --with-lid-opt="/path/to/foo.o /path/to/bar.o" ...
```

如果有很多 .o 文件，最好打包到一个静态库里，例如：

```
ar rcus libmyluafiles.a *.o
```

然后把 myluafiles 包连接到 Nginx 运行文件中：

```
./configure \
    --with-ld-opt="-L/path/to/lib -Wl,--whole-archive -lmyluafiles -Wl,--no-
whole-archive"
```

/path/to/lib 是包含 libmyluafiles.a 文件的路径。这里要用到 --whole-archive 选项，否则库会因为缺少符号被跳过。

## 10. 在 Nginx 工作进程中共享数据

把共享数据封装进一个 Lua 模块，以便于在 Nginx 工作进程处理的所有请求中共享全局数据。子请求中使用 require 连接模块，然后在 Lua 里操作共享数据。因为需要 Lua 模块装载一次，然后所有的线程（例程）共享相同的模块副本（代码和数据）。注意：因为单线程处理单个请求这种隔离设计会导致 Lua 全局变量（非模块级别变量）不会在请求间持续存在。

示例：

```
-- mydata.lua
local _M = {}

local data = {
    dog = 3,
    cat = 4,
    pig = 5,
}

function _M.get_age(name)
    return data[name]
end

return _M
```

在 nginx.conf 中访问：

```
location /lua {
    content_by_lua_block {
        local mydata = require "mydata"
        ngx.say(mydata.get_age("dog"))
    }
}
```

示例中的 mydata 模块只会被第一个 /lua 请求装载和运行，相同工作进程中所有的子请求将使用重新载入的模块实例，数据的副本完全相同，直到 HUP 信号被发送给 Nginx 管理进程去强制重新载入配置。这个数据分享技术在高性能 Lua 应用程序上是非常有必要的。

这个数据分享是基于工作线程的，而非基于服务的，即不能跨工作进程在整个服务上有效。同一个 Nginx 管理进程下的多个工作进程不能使用这项技术实现跨进程的股份。



通常推荐在共享只读数据的情况下使用这项技术。可以在每个工程进程的子线程请求中共享可改变数据，但要保证在这段时间内没有非阻塞的 I/O 操作（包括 ngx.sleep）。这段时间内，不能返回 Nginx 的消息循环和 ngx\_lua 的轻线程调度，它们不会让彼此处于竞争情况。因为这个原因，对于工作进程级别的可变数据共享需要非常小心。

如果需要服务器级的数据共享，需要使用下面一项或多项技术。

- 1) 使用 ngx.shared.DICT API。
- 2) 使用只有一个工作进程的单 Nginx 服务（不推荐这种模式，因为现在的服务器基本上是多处理器或单处理器多核心）。
- 3) 使用数据存储机制，如 Memcached、Redis、MySQL 或 PostgreSQL。OpenResty 工具平台绑定了很多 Nginx 模块和 Lua 库，提供了数据访问机制。

## 11. TCP 套接字连接操作

tcpsock::connect 方法可以在连接被拒绝等错误情况下一律返回成功标识，然而下步操作 cosocket 对象将失败，返回连接实际的错误信息。

这个问题由 Nginx 的消息模型限制引起，并且只在 Mac OS X 上发生。

## 12. Lua 线程的挂起和恢复

Lua 中的 dofile 和 require 在 Lua5.1 和 LuaJIT2.0/2.1 中作为 C 函数内置，当通过 dofile 或 require 载入的 Lua 文件调用 ngx.location.capture\*、ngx.exec、ngx.exit 或其他 API 时，会挂起最高级别的 Lua 文件，进而引发 “attempt to yield across C-call boundary” 这个错误。为避免这个问题，需要把会引起挂起的调用放到自己的 Lua 文件中，以函数形式存在，不要把这些调用放到库里面。

因为标准的 Lua5.1 虚拟机不支持完全恢复操作，方法 ngx.location.capture、ngx.location.capture\_multi、ngx.redirect、ngx.exec 和 ngx.exit 不能在 pcall() 或 xpcall 或者在 for ... in ... 第一行中被使用，否则会引发 “a ttempt to yields across metamethod/C-call boundary” 错误。可使用支持完整恢复机制的 LuaJIT 2.X 避免这个问题。

## 13. Lua 变量范围

导入模块的时候要使用规范做法：

```
local xxx = require('xxx')
```

不要使用已经过时的做法：

```
require('xxx')
```

这样做的原因：设计上，全局变量拥有与之关联 Nginx 请求处理器相同的生命周期。每一个请求处理器拥有自己的一系列全局变量，并且请求之间是隔离的。Lua 模块被第一个 Nginx 请求处理器载入，并且被 require() 缓存在 package.loaded 表，后续引用都将使用这个实例。内置的 module() 被相同的 Lua 模块使用，设置一个已经载入模块表中的全局变量会

产生副作用，但是这个全局变量会由请求处理者在最后清理，每一个子请求处理者都拥有它们自己的全局环境，所以有可能访问到一个已经被清理的全局变量，而这个 nil 值会产生 Lua 异常，所以推荐使用新方法。

通常在 ngx\_lua 上下文中使用 Lua 全局变量是一个非常糟糕的主意。

- 1) 在协程请求上滥用实际上应该是本地变量的全局变量有非常多的副作用。
- 2) 全局变量需要在全局环境下查表，价格高昂。
- 3) 一些全局变量引用会引起很多打印错误，这将很影响调试。

推荐通过 local 声明在本范围使用的变量。

要找出所有在 Lua 代码中使用的 Lua 全局变量，可以使用 lua-releng 工具：

```
$ lua-releng
Checking use of Lua global variables in file lib/foo/bar.lua ...
      1      [1489]  SETGLOBAL          7 -1      ; contains
     55      [1506]  GETGLOBAL          7 -3      ; setvar
      3      [1545]  GETGLOBAL          3 -4      ; varexpend
```

输出显示在 lib/foo/bar.lua 的 1489 行写入了 contains 变量，在 1506 行读取了 setvar 变量，在 1545 行读取了全局的 varexpend 变量。

这个工具保证本地变量全部被 local 关键字声明，否则会抛出一个运行期异常，从而避免访问这些变量引起的资源竞争。

#### 14. location 访问的局限

ngx.location.capture 和 ngx.location.capture\_multi 配置项不能捕捉包含 add\_before\_body、add\_after\_body、auth\_request、echo\_location、echo\_location\_async、echo\_subrequest 或 echo\_subrequest\_async 这些配置项的 location。

```
location /foo {
    content_by_lua_block {
        res = ngx.location.capture("/bar")
    }
}
location /bar {
    echo_location /blah;
}
location /blah {
    echo "Success!";
}
```

```
$ curl -i http://example.com/foo
```

因为 Location/bar 里包含了 echo\_location，所以示例并不能按预期工作。

#### 15. CoSocket 应用限制

cosocket 并非在任何地方都有效。因为 Nginx 核心内部的限制，cosocket API 在下面的

上文中被禁用: `set_by_lua*`、`log_by_lua*`、`header_filter_by_lua*`、`body_filter_by_lua`。

`cosocket` 通常在 `init_by_lua*` 和 `init_work_by_lua*` 配置项中被禁用,但在未来这个功能可以加上,因为实际上 Nginx 核心并没有限制。

当原始的上下文不需要等待 `cosocket` 结果时,通过 `ngx.timer.at` 创建一个零延迟的时钟,在时钟回调中异步地处理 `cosocket` 结果,即可使用 `cosocket`。

## 16. 特殊的转义序列

因为 Lua 语言解析器和 Nginx 配置文件解析器在处理之前会将 `\` 去掉,所以避免在正则规则中使用 `\d`、`\s` 或 `\w` 这种转义:

```
# nginx.conf
? location /test {
?     content_by_lua '
?         local regex = "\d+" -- THIS IS WRONG!!
?         local m = ngx.re.match("hello, 1234", regex)
?         if m then ngx.say(m[0]) else ngx.say("not matched!") end
?     ';
? }
# evaluates to "not matched!"
```

避免使用双 `\`:

```
# nginx.conf
location /test {
    content_by_lua '
        local regex = "\\d+"
        local m = ngx.re.match("hello, 1234", regex)
        if m then ngx.say(m[0]) else ngx.say("not matched!") end
    ';
}
# evaluates to "1234"
```

Nginx 配置文件会将 `\\d+` 解析为 `\d+`,然后由 Lua 解析器解析为 `\d+`。

另一种方法,正则表达规则可以放到 `[[...]]` 中,只会被 Nginx 配置文件过滤一次,这是 Lua 长字符串的一种格式:

```
# nginx.conf
location /test {
    content_by_lua '
        local regex = [[\d+]]
        local m = ngx.re.match("hello, 1234", regex)
        if m then ngx.say(m[0]) else ngx.say("not matched!") end
    ';
}
# evaluates to "1234"
```

`[[\d+]]` 被 Nginx 配置文件解析成了 `[[\d+]]`,而这是预期的,是正确的。

长的正则规则中可能包含 `[...]`,形成 `[=[...]=]` 这样的规则。

```
# nginx.conf
location /test {
    content_by_lua '
        local regex = [=[[0-9]+=]
        local m = ngx.re.match("hello, 1234", regex)
        if m then ngx.say(m[0]) else ngx.say("not matched!") end
    ';
}
```

# evaluates to "1234"

另外一个选择是把 Lua 代码放到一个外部的脚本文件中，通过 `*_by_lua_file` 调用，这样，转义符就只会被过滤一次。

```
-- test.lua
local regex = "\\d+"
local m = ngx.re.match("hello, 1234", regex)
if m then ngx.say(m[0]) else ngx.say("not matched!") end
-- evaluates to "1234"
```

同样，括号内的转义符也需要修正。Lua 不对 [...] 中的符号做修改，直接使用正则规则即可。

```
-- test.lua
local regex = [[\d+]]
local m = ngx.re.match("hello, 1234", regex)
if m then ngx.say(m[0]) else ngx.say("not matched!") end
-- evaluates to "1234"
```

## 17. 不支持 SSI 混合模式

不支持在相同的 Nginx 请求中使用 `ngx_lua` 进行 SSI 混合，只能完全通过 `ngx_lua` 自己实现 SSI 的功能，这更有效。

## 18. 不完全支持 SPDY 模式

当前 `ngx_lua` 提供的 Lua API 不支持 Nginx SPDY 模式：`ngx.location.capture`、`ngx.location.capture_multi`、`ngx.`

## 27.2 Lua 配置顺序

Nginx 中的 Lua 代码用配置项分隔成一个个脚本块，这些配置项决定 Lua 代码何时运行以及何时结束被使用。关于 Lua 在 Nginx 下的运行机制以及 Nginx 在 HTTP 处理过程中的阶段机制前文已经有讲述，图 27-1 再次描述 Lua 代码可以运行的阶段，用于参照研读下面的配置指令。

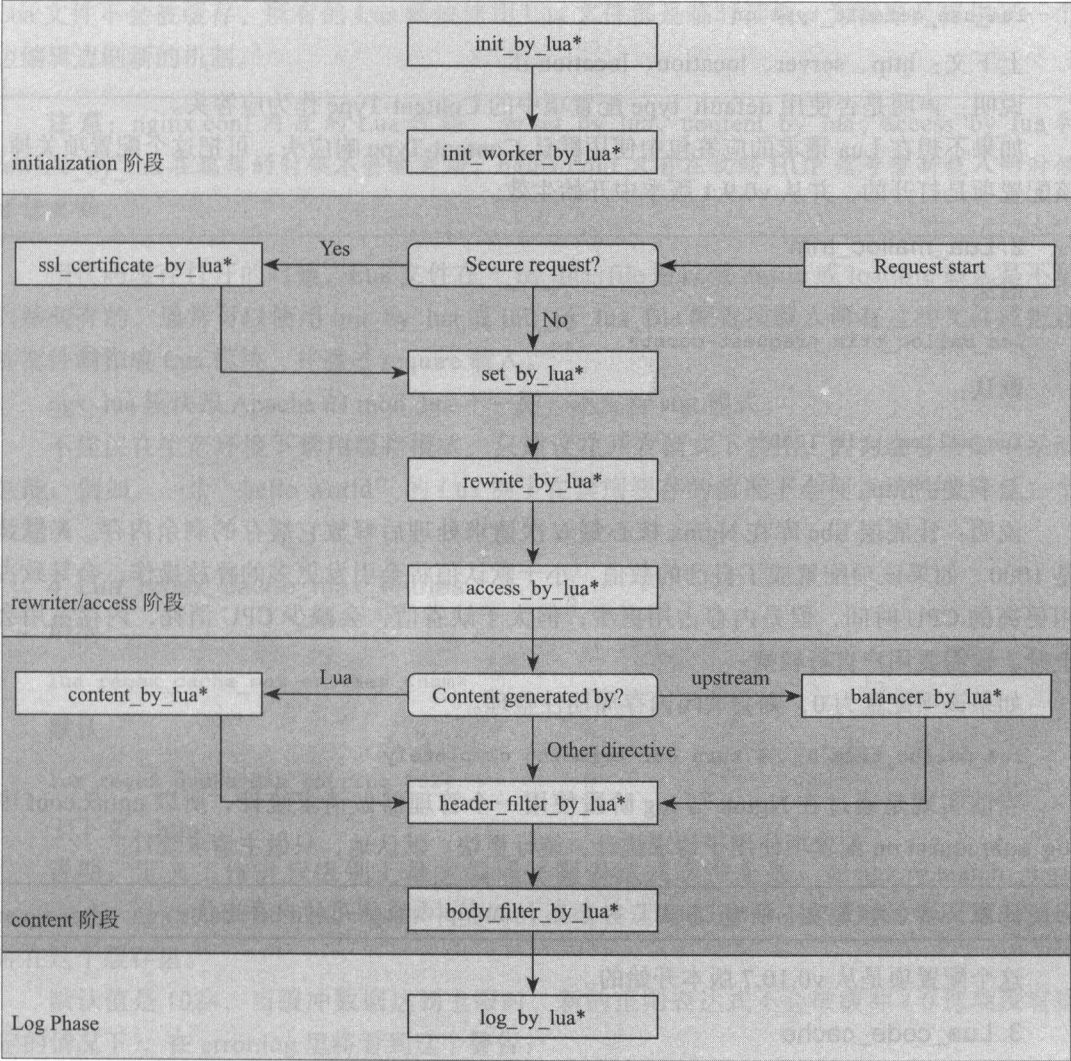


图 27-1 Lua 模块指令运行顺序

## 27.3 配置指令

下面详细描述 ngx\_lua 中的配置指令。

### 1. lua\_use1\_default\_type

语法：

lua\_use\_default\_type on | off

默认：

```
lua_use_default_type on
```

上下文: http、server、location、location if。

说明: 声明是否使用 default\_type 配置项中的 Content-Type 作为应答头。

如果不想在 Lua 请求的应答包中使用默认 Content-Type 响应头, 可把这个配置项关掉。该配置项是打开的, 并从 v0.9.1 版本中开始生效。

## 2. Lua\_malloc\_trim

语法:

```
lua_malloc_trim <request-count>
```

默认:

```
lua_malloc_trim 1000
```

上下文: http。

说明: 让底层 libc 库在 Nginx 核心每  $N$  次请求处理后释放它缓存的剩余内存。 $N$  默认是 1000。如果该项配置成了自己的数值, 小于默认值将会引发更多的释放操作, 会导致占用更高的 CPU 时间, 但是内存占用更少; 但大于缺省值, 会减少 CPU 消耗, 内存占用会上升, 这需要用户自行权衡。

如果该项配置为 0, 则会关闭内存周期性整理。

```
lua_malloc_trim 0; # turn off trimming completely
```

当前实现是通过在 Nginx 写 log 阶段使用一个管理器做请求统计, 所以 nginx.conf 中 log\_subrequest on 配置项使用子请求统计, 统计更快。默认地, 只做主请求统计。

---

**注意:** 这个配置项不影响 LuaJIT 本身基于 mmap 系统调用的内存申请。

---

这个配置项是从 v0.10.7 版本开始的。

## 3. Lua\_code\_cache

语法:

```
lua_code_cache on | off
```

默认:

```
lua_code_cache on
```

上下文: http、server、location、location if。

说明: 在 \*\_by\_lua\_file 和 Lua 模块配置项里缓冲 Lua 代码 (如 set\_by\_lua\_file 和 content\_by\_lua\_file)。

从 0.9.3 版本开始, 当配置项为 off 时, 每一个 ngx\_lua 的请求服务将在一个独立的 Lua VM 实例中处理, 所以被 set\_by\_lua\_file、content\_by\_lua\_file、access\_by\_lua\_file 引用的

Lua 文件不会被缓存，所有的 Lua 模块使用 Lua 文件都是临时读取，为程序员提供了一个边编辑边刷新的机制。

**注意：**nginx.conf 内嵌的 Lua 代码，如 `set_by_lua`、`content_by_lua`、`access_by_lua` 和 `rewrite_by_lua` 在编辑的时候不会被更新，nginx.conf 只有在收到 HUP 信号重新载入的时候才能更新。

当代码缓存打开的时候，Lua 文件在 `*_by_lua_file` 阶段被 `dofile` 或 `loadfile` 载入是不能被缓存的，通常可以使用 `init_by_lua` 或 `init_by_lua_file` 配置项载入所有这些文件或把这些文件制作成 Lua 模块，并通过 `require` 载入。

ngx\_lua 模块跟 Apache 的 `mod_lua` 不一样，不支持 `stat` 模式。

不建议在生产环境下禁用缓存模式，只建议在开发模式下禁用，因为会非常影响系统性能。例如，一个“hello world”的 Lua 例子在禁用缓存的情况下会使系统性能降低一个数量级。

#### 4. Lua\_regex\_cache\_max\_entries

语法：

```
lua_regex_cache_max_entries <num>
```

默认：

```
lua_regex_cache_max_entries 1024
```

上下文：http。

说明：定义工作进程级别上最大编译正则表达式缓冲条数。在 `ngx.re.match`、`ngx.re.gmatch`、`ngx.re.sub` 和 `ngx.re.gsub` 中使用的正则表达式在选项 `o` 被指定的情况下会被缓冲在这个缓存里。

默认值是 1024，当缓冲数据达到上限时，新的正则表达式不会被缓冲（0 选项没有指定的情况下），在 `error.log` 里将看到这个警告：

```
2011/08/27 23:18:26 [warn] 31997#0: *1 lua exceeding regex cache max entries (1024), ...
```

如果正在使用通过 `resty.core.regex` 载入 `lua-resty-core` 模块使用 `ngx.re.*API`，那么，会使用一个 LRU 缓冲管理算法管理缓存。

使用正则表达式的时候不要激活 0 选项（或使用 `replace` 来修改 `ngx.re.sub` 和 `ngx.re.gsub` 参数），这可以保证避免达到数量限制。

#### 5. Lua\_regex\_match\_limit

语法：

```
lua_regex_match_limit <num>
```



默认:

```
lua_regex_match_limit 0
```

上下文: http。

说明: 指定使用 ngx.re API 时 PCRE 库用的 “match limit”。

PCRE 手册上指出, 这个限制影响回溯发生的数量。当这个限制命中的时候, Lua 侧的 ngx.re API 会收到 “pcre\_exec() failed: -8” 的错误字符串。当限制设置为 0 的时候, 会使用编译时设置的默认值, 这也是这个配置项的默认值。

## 6. lua\_package\_path

语法:

```
lua_package_path <lua-style-path-str>
```

默认: LUA\_PATH 环境变量或编译时内建默认值。

上下文: http。

说明: 设置 set\_by\_lua、content\_by\_lua 和其他配置项使用到的 Lua 模块搜索路径。路径字符串使用 Lua 路径格式, ;; 可以用来表示原来的搜索路径。

从 v0.5.0rc29 版本开始, 可以在搜索路径中使用 \$prefix 或 \${prefix}, 用以指明 server 前缀。这个前缀是 Nginx 服务启动时通过 -p PATH 命令行传进来的。

## 7. lua\_package\_cpath

语法:

```
lua_package_cpath <lua-style-cpath-str>
```

默认: LUA\_PATH 环境变量或编译时内建默认值。

上下文: http。

说明: 设置 set\_by\_lua、content\_by\_lua 和其他配置项脚本中用到的 Lua C 模块的搜索路径。这也是使用 Lua 路径格式, 可以用 ;; 表示原有路径。

## 8. init\_by\_lua

语法:

```
init_by_lua <lua-script-str>
```

上下文: http。

阶段: loading-config。

说明: 从 v0.9.17 版本开始, 不鼓励使用这个配置项了, 推荐使用 init\_by\_lua\_block。

当 Nginx 主进程载入 Nginx 配置文件的时候, 将在全局的 Lua VM 级别运行 <lua-script-str>。

当 Nginx 收到 HUP 信号重新载入配置文件的时候, Lua VM 自动重新创建, init\_by\_lua 将再次在新的 Lua VM 中运行。当 lua\_code\_cache 配置项关闭的时候, init\_by\_lua 管理



者将会在每一个请求中运行一个标准的 Lua VM (虚拟机)。

通常可以在服务启动阶段使用这个配置项注册全局变量或预加载 Lua 模块, 下面是一个预加载模块的例子:

```
init_by_lua 'cjson = require "cjson";

server {
    location = /api {
        content_by_lua_block {
            ngx.say(cjson.encode({dog = 5, cat = 6}))
        }
    }
}
```

也可以在这个阶段初始化 lua\_shared\_dict 共享内存。例如:

```
lua_shared_dict dogs 1m;

init_by_lua '
    local dogs = ngx.shared.dogs;
    dogs:set("Tom", 56)
';

server {
    location = /api {
        content_by_lua_block {
            local dogs = ngx.shared.dogs;
            ngx.say(dogs:get("Tom"))
        }
    }
}
```

需要注意的是, 配置文件重载不能清除 lua\_shared\_dict 的共享内存, 所以在这种情况下, 需要在 init\_by\_lua 代码中重新初始化共享内存, 需要设置一个自定义标志, init\_by\_lua 代码总是需要检查这个标志。因为这段 Lua 代码在 Nginx 工作进程运行前运行, 数据和代码在这里享受写时复制 copy-on-write (COW) 性能, 这是许多操作系统为工作进程提供的机制, 这将节约很多内存。

不要在这里初始化自己的全局变量, 因为使用全局变量容易出错, 还可能污染命名空间。推荐使用 Lua 模块文件 (但是不要使用标准 Lua 函数的 module 定义模块, 因为它会污染命名空间), 并且在 init\_by\_lua 或其他步骤调用 require() 装载自己的模块文件 (require 不在 package.loaded 缓存载入模块, 所以模块只能在整个 Lua VM 实例中被载入一次)。

这个阶段只有少数的 Nginx Lua API 可用:

- 日志 API: ngx.log 和 print。
- 共享内存 API: ngx.shared.DICT。

更多的 Nginx API 将在后面的阶段支持。

基本上, 在这个阶段可以安全地使用 Lua 库做阻塞 I/O 操作, 因为在服务启动阶段是

可以阻塞管理进程的。每一个 Nginx 核心在配置载入阶段执行阻塞 I/O（至少解析上游服务器主机名是阻塞的）。

必须注意本阶段 Lua 的安全性，因为 Nginx 管理进程通常在 root 账户下运行。

### 9. init\_by\_lua\_block

语法：

```
init_by_lua_block { lua-script }
```

上下文：http。

阶段：loading-config。

说明：跟 init\_by\_lua 配置项类似，除了这个配置项直接在 {} 里使用内嵌 Lua 代码，代替原来的'' 分隔。例如：

```
init_by_lua_block {
    print("I need no extra escaping here, for example: \r\nblah")
}
```

### 10. init\_by\_lua\_file

语法：

```
init_by_lua_file <path-to-lua-script-file>
```

上下文：http。

阶段：loading-config。

说明：与 init\_by\_lua 一样，只是 Lua 代码是放在文件内的。

如果指定了类似 foo/bar.lua 这样的路径，将会使用启动 Nginx 时的命令行中的 -p PATH 选项传进来的 server prefix，使用它转换为绝对路径。

### 11. init\_worker\_by\_lua

语法：

```
init_worker_by_lua <lua-script-str>
```

上下文：http。

阶段：starting-worker。

说明：从 v0.9.17 版本开始，这个配置项就不再鼓励使用了，推荐使用新的配置项 init\_worker\_by\_lua\_block。

当管理进程已经启动并开始工作了，管理进程启动每一个工作进程，工作进程启动时指定的 Lua 代码就会被运行。如果管理进程还没有启动，需要做一些处理，需要在 init\_by\_lua\* 中注册代码。

这个阶段通常用于创建基于工作进程的重复时钟（通过 ngx.timer.at API），进行后端的健康检查或其他时间周期性工作，下面是一个例子。

```

init_worker_by_lua '
    local delay = 3 -- in seconds
    local new_timer = ngx.timer.at
    local log = ngx.log
    local ERR = ngx.ERR
    local check

    check = function(premature)
        if not premature then
            -- do the health check or other routine work
            local ok, err = new_timer(delay, check)
            if not ok then
                log(ERR, "failed to create timer: ", err)
                return
            end
        end
    end

    local ok, err = new_timer(delay, check)
    if not ok then
        log(ERR, "failed to create timer: ", err)
        return
    end
';

```

## 12. init\_worker\_by\_lua\_block

语法:

```
init_worker_by_lua_block { lua-script }
```

上下文: http。

阶段: starting-worker。

说明: 与 init\_worker\_by\_lua 不同的是使用 {} 代替 Nginx 字体串内嵌代码, 其他都与 init\_worker\_by\_lua 相同。例如:

```

init_worker_by_lua_block {
    print("I need no extra escaping here, for example: \r\nblah")
}

```

## 13. init\_worker\_by\_lua\_file

语法:

```
init_worker_by_lua_file <lua-file-path>
```

上下文: http。

阶段: starting-worker。

说明: 跟 init\_worker\_by\_lua 相比, 除了把代码从 {} 移到 Lua 源文件或字节码文件外, 其他都是一样的。

## 14. set\_by\_lua

语法:

```
set_by_lua $res <lua-script-str> [$arg1 $arg2 ...]
```

上下文: server、server if、location、location if。

阶段: rewrite。

说明: 运行 <lua-script-str> 指定的代码, \$arg1 \$arg2 ... 是输入参数, \$res 是输出返回字符串。<lua-script-str> 中的代码可以调用 API, 可以从 ngx.arg 表 (索引从 1 开始顺序增长) 获取输入参数。

因为 Nginx 事件循环在 <lua-script-str> 代码运行阶段是阻塞的, 所以配置项被设计成运行短、快的代码块, 应该避免运行消耗时间的代码。

这个配置项用于向 ngx\_http\_rewrite\_module 标准命令列表中插入自定义命令。因为 ngx\_http\_rewrite\_module 自己的命令不支持非阻塞 I/O, 所以不能在这个配置项内运行需要挂起当前 Lua 轻线程 (协程) 类的 API。

至少下列 API 和函数在 set\_by\_lua 中被禁用:

- 输出类 API, 如 ngx.say 和 ngx.send\_headers。
- 控制类 API, 如 ngx.exit。
- 子请求类 API, 如 ngx.location.capture 和 ngx.location.capture\_multi。
- cosocket 类 API, 如 ngx.socket.tcp 和 ngx.req.socket。
- 睡眠类 API, 如 ngx.sleep。

另外, 同一时间内, 这个配置项只能写并输出一个 Nginx 变量。可以使用 ngx.var.VARIABLE。

```
location /foo {
    set $diff ''; # 定义 $diff 变量

    set_by_lua $sum '
        local a = 32
        local b = 56

        ngx.var.diff = a - b; -- 直接写入 $diff
        return a + b;         -- 正常地返回 $sum 值
    ';

    echo "sum = $sum, diff = $diff";
}
```

set\_by\_lua 配置项可以自由地和 ngx\_http\_rewrite\_module、set\_misc\_nginx\_module、array\_var\_nginx\_module 模块的配置项混合使用。所有这些配置项会根据它们在配置文件中的顺序运行。

```
set $foo 32;
```

```
set_by_lua $bar 'return tonumber(ngx.var.foo) + 1';
set $baz "bar: $bar"; # $baz == "bar: 33"
```

从 v0.5.0rc29 版本开始, <lua-script-str> 禁止插入 Nginx 变量。配置项中可以直接使用 \$ 符号。

这个配置项需要 ngx\_devel\_kit 模块。

### 15. set\_by\_lua\_block

语法:

```
set_by_lua_block $res { lua-script }
```

上下文: server、server if、location、location if。

阶段: rewrite。

说明: 跟 set\_by\_lua 相同, 除了以下两点。

- Lua 代码在 {} 中, 取代了 Nginx 字符串语义。
- 不支持额外的参数。

例如:

```
set_by_lua_block $res { return 32 + math.cos(32) }
# $res now has the value "32.834223360507" or alike.
```

代码中不需要转义。

### 16. set\_by\_lua\_file

语法:

```
set_by_lua_file $res <path-to-lua-script-file> [$arg1 $arg2 ...]
```

上下文: server、server if、location、location if。

阶段: rewrite。

说明: 除了代码放在 <path-to-lua-script-file> 中, 其他均与 set\_by\_lua 相同, 从 v0.5.0rc32 版本开始, 支持 Lua/LuaJIT 字节码。

<path-to-lua-script-file> 支持变量插入。需要注意的是, 在处理变量时要特别小心, 防止注入攻击。

当使用 foo/bar.lua 类的路径时, 模块会使用启动 Nginx 命令行时的 -p PATH 选项确定的 server prefix, 从而将路径转换成绝对路径。

Lua 代码缓存打开的时候 (默认是打开的), 用户代码会在第一个请求到来的时候被装载, 并被缓存起来, 每次代码编辑过后, Nginx 必须重新载入配置文件。Lua 代码缓存可以在开发阶段临时关闭, 把 lua\_code\_cache 设置为 off 可以避免重载 Nginx。

### 17. content\_by\_lua

语法:

```
content_by_lua <lua-script-str>
```

上下文: location、location if。

阶段: content。

说明: 从 v0.9.17 版本开始, 不鼓励使用这个配置项了, 推荐使用 `content_by_lua_block`。

每一个请求到来的时候, `<lua-script-str>` 中的 Lua 代码都会被运行。Lua 代码可以调用 API, 它作为一个新创建的协程在独立的全局环境中运行 (像一个沙箱)。

不要在相同的 location 中同时使用本配置项和其他内容管理器, 例如, 本配置项不能和 `proxy_pass` 配置项在相同的 location 中使用。

### 18. content\_by\_lua\_block

语法:

```
content_by_lua_block { lua-script }
```

上下文: location、location if。

阶段: content。

说明: 除了代码放在 `{}` 中, 其他均与 `content_by_lua` 相同。例如:

```
content_by_lua_block {
    ngx.say("I need no extra escaping here, for example: \r\nblah")
}
```

### 19. content\_by\_lua\_file

语法:

```
content_by_lua_file <path-to-lua-script-file>
```

上下文: location、location if。

阶段: content。

说明: 除了代码放在文件中, 其他均与 `content_by_lua` 相同。从 v0.5.0rc32 版本开始, 支持 Lua/LuaJIT 字节码运行。

可以在 `<path-to-lua-script-file>` 中使用 Nginx 的变量, 当然这样做会引入一些风险, 所以不推荐这种做法。

当使用 `foo/bar.lua` 类型的路径时, 模块会使用 Nginx 启动时的 `-p PATH` 参数设置的 `server prefix` 路径参数, 从而将路径转换成绝对路径。

同样地, Lua 代码缓存也会影响调试, 可参见其他配置项相同部分。

Nginx 变量支持在文件路径中的动态匹配, 例如:

```
# WARNING: Nginx 变量中的内容必须仔细过滤, 否则, 将会是巨大的安全隐患。
location ~ ^/app/([_a-zA-Z0-9/]+) {
    set $path $1;
```

```

    content_by_lua_file /path/to/lua/app/root/$path.lua;
}

```

必须仔细校验和过滤输入的参数，以阻止恶意的用户和访问。

## 20. rewrite\_by\_lua

语法：

```
rewrite_by_lua <lua-script-str>
```

上下文：http、server、location、location if。

阶段：rewrite tail。

说明：从 v0.9.17 版本开始，不鼓励使用这个配置项了，推荐使用 `rewrite_by_lua_block`。

每一个请求都会引发 `<lua-script-str>` 中的 Lua 代码运行。Lua 代码可以调用 API，在一个独立的全局环境里作为一个新的协程运行。

注意，这个管理器总是在 `ngx_http_rewrite_module` 之后运行，所以下面的代码将会按预期工作：

```

location /foo {
    set $a 12; # create and initialize $a
    set $b ""; # create and initialize $b
    rewrite_by_lua 'ngx.var.b = tonumber(ngx.var.a) + 1';
    echo "res = $b";
}

```

因为 `set $a 12` 和 `set $b ""` 在 `rewrite_by_lua` 之前运行。

下面的例子中，代码将不会按预期运行：

```

? location /foo {
?     set $a 12; # create and initialize $a
?     set $b ''; # create and initialize $b
?     rewrite_by_lua 'ngx.var.b = tonumber(ngx.var.a) + 1';
?     if ($b = '13') {
?         rewrite ^ /bar redirect;
?         break;
?     }
?     echo "res = $b";
? }

```

因为 `if` 在 `rewrite_by_lua` 事件之前运行，如果 `if` 放在 `rewrite_by_lua` 之后，正确的代码应该是这样的：

```

location /foo {
    set $a 12; # create and initialize $a
    set $b ''; # create and initialize $b
    rewrite_by_lua '
        ngx.var.b = tonumber(ngx.var.a) + 1
        if tonumber(ngx.var.b) == 13 then

```

```

        return ngx.redirect("/bar");
    end
';

    echo "res = $b";
}

```

注意，ngx\_eval 模块可以使用 rewrite\_by\_lua 实现，例如：

```

location / {
    eval $res {
        proxy_pass http://foo.com/check-spam;
    }

    if ($res = 'spam') {
        rewrite ^ /terms-of-use.html redirect;
    }

    fastcgi_pass ...;
}

```

用 ngx\_lua 实现：

```

location = /check-spam {
    internal;
    proxy_pass http://foo.com/check-spam;
}

location / {
    rewrite_by_lua '
        local res = ngx.location.capture("/check-spam")
        if res.body == "spam" then
            return ngx.redirect("/terms-of-use.html")
        end
    ';

    fastcgi_pass ...;
}

```

和其他 rewrite 过程管理器一样，rewrite\_by\_lua 总是在子请求中运行。

注意，当在 rewrite\_by\_lua 管理器中调用 ngx.exit (ngx.OK) 时，Nginx 请求处理控制流将仍然继续运行内容管理器。要想中断当前 rewrite\_by\_lua 管理器，需要使用 status>=200 (ngx.HTTP\_OK) 并且通过 status<300 (ngx.HTTP\_SPECIAL\_RESPONSE) 调用 ngx.exit，可以成功退出。ngx.exit (ngx.HTTP\_INTERNAL\_SERVER\_ERROR) (或其错误码族) 用于表示失败退出。

ngx\_http\_rewrite\_module 的重写配置项被用来改变 URI、初始化内部定向操作，如果使用了 ngx\_http\_rewrite\_module，则情况会变得复杂。例如：

```

location /foo {
    rewrite ^ /bar;
    rewrite_by_lua 'ngx.exit(503)';
}

```



```
location /bar {
    ...
}
```

ngx.exit (503) 不会运行, rewrite ^ /bar 在这里初始化了一个内部重定向, 所以 rewrite\_by\_lua 不会运行。

rewrite\_by\_lua 代码总是在 rewrite 请求预处理过程之后运行, 除非 rewrite\_by\_lua\_no\_postpone 打开。

## 21. rewrite\_by\_lua\_block

语法:

```
rewrite_by_lua_block { lua-script }
```

上下文: http、server、location、location if。

阶段: rewrite tail。

说明: 除了代码放在 {} 中, 其他均和 rewrite\_by\_lua 相同。例如:

```
rewrite_by_lua_block {
    do_something("hello, world!\nhiya\n")
}
```

## 22. rewrite\_by\_lua\_file

语法:

```
rewrite_by_lua_file <path-to-lua-script-file>
```

上下文: http、server、location、location if。

阶段: rewrite tail。

说明: 和 rewrite\_by\_lua 的区别是代码放在 <path-to-lua-script-file> 中, 并且从 v0.5.0rc32 版本开始, 支持字节码。或在源码文件中使用 Nginx 变量, 但是要防范注入等风险, 从安全性角度考虑, 不是推荐这种做法。同样地, 当文件名是 foo/bar.lua 类似的风格时, 模块会使用 server prefix 转换为绝对路径。Lua 代码缓存也会让代码在第一个请求时被载入。开发阶段想要快速修改代码调试程序, 需要临时把 lua\_code\_cache 从默认打开改成关闭, 否则就需要每次重新载入 nginx.conf 文件。

rewrite\_by\_lua\_file 代码总是在 rewrite 请求预处理阶段之后运行, 除非 rewrite\_by\_lua\_no\_postpone 打开。

## 23. access\_by\_lua

语法:

```
access_by_lua <lua-script-str>
```

上下文: http、server、location、location if。

阶段: access tail。

说明：从 v0.9.17 版本，推荐使用 `access_by_lua_block` 代替。

在每一个请求来的时候，`<lua-script-str>` 的 Lua 代码被运行，作为访问阶段的管理者。Lua 脚本可以进行 API 调用，会在一个独立的全局环境中新建一个协程运行，类似沙箱。

注意，本管理器在标准的 `ngx_http_access_module` 之后运行，所以，下面的代码将按预期运行。

```
location / {
    deny    192.168.1.1;
    allow   192.168.1.0/24;
    allow   10.1.1.0/16;
    deny    all;

    access_by_lua '
        local res = ngx.location.capture("/mysql", { ... })
        ...
    ';

    # proxy_pass/fastcgi_pass/...
}
```

这里，如果一个客户端 IP 在黑名单中，将被拒绝往下执行，不会执行 MySQL 数据库查询。

注意，`ngx_auth_request` 模块可以用 `access_by_lua` 实现：

```
location / {
    auth_request /auth;

    # proxy_pass/fastcgi_pass/postgres_pass/...
}
```

还可以用 `ngx_lua` 实现：

```
location / {
    access_by_lua '
        local res = ngx.location.capture("/auth")

        if res.status == ngx.HTTP_OK then
            return
        end

        if res.status == ngx.HTTP_FORBIDDEN then
            ngx.exit(res.status)
        end

        ngx.exit(ngx.HTTP_INTERNAL_SERVER_ERROR)
    ';

    # proxy_pass/fastcgi_pass/postgres_pass/...
}
```

和其他的访问阶段（access）管理者一样，`access_by_lua` 不能在子请求中工作。

在 `access_by_lua` 管理中调用 `ngx.exit(ngx.OK)` 时，Nginx 请求控制流程还会继续

运行管理器。要在当前的请求处理管理器中断处理，要用错误码为 200 ~ 300 的值，`status>=200 (ngx.HTTP_OK)` 和 `status<300 (ngx.HTTP_SPECIAL_RESPONSE)` 表示成功退出，`ngx.exit (ngx.HTTP_INTERNAL_SERVER_ERROR)`(或其友类值) 表示失败退出。

从 v0.9.20 版本开始，可以使用 `access_by_lua_no_postpone` 配置项控制使用本管理器代替 `access` 的请求管理阶段工作。

## 24. access\_by\_lua\_block

语法：

```
access_by_lua_block { lua-script }
```

上下文：http、server、location、location if。

阶段：access tail。

说明：与 `access_by_lua` 不同的仅仅是代码放在 {} 中。例如：

```
access_by_lua_block {
    do_something("hello, world!\nhiya\n")
}
```

## 25. access\_by\_lua\_file

语法：

```
access_by_lua_file <path-to-lua-script-file>
```

上下文：http、server、location、location if。

阶段：access tail。

说明：与 `access_by_lua` 不同的是代码放在 `<path-to-lua-script-file>` 指定的文件里，同时从 v0.5.0rc32 版本开始，支持 Lua/LuaJIT 字节码。

可以在 Lua 文件中使用 Nginx 变量实现复杂的功能，但是考虑到风险的因素，不推荐这样做。

当指定类似于 `foo/bar.lua` 的文件名时，模块会进行绝对路径的转换，使用启动 Nginx 时命令行通过 `-p PATH` 选项传入的路径设置的 `server prefix` 进行转换。同样，因为默认的 Lua 代码缓存是打开的，代码编辑后需要重载服务器的 `nginx.conf` 文件才能生效，为了方便调试代码，可以临时将 `lua_code_cached` 关闭，避免每次都要重新装载配置文件。

通过文件路径传入的 Nginx 变量动态分配与 `content_by_lua_file` 一样。

## 26. header\_filter\_by\_lua

语法：

```
header_filter_by_lua <lua-script-str>
```

上下文：http、server、location、location if。

过程：output-header-filter。

说明：从 v0.9.17 版本开始，推荐使用 `header_filter_by_lua_block` 代替。使用 `<lua-script-str>` 指定的 Lua 代码定义一个输出头过滤器。

注意，下面的 API 在这里不能使用：

- 输出类 API，如 `ngx.say` 和 `ngx.send_headers`。
- 控制类 API，如 `ngx.redirect` 和 `ngx.exec`。
- 子请求类 API，如 `ngx.location.capture` 和 `ngx.location.capture_multi`。
- `cosocket` 类 API，如 `ngx.socket.tcp` 和 `ngx.req.socket`。

下面是一个修改应答头的例子：

```
location / {
    proxy_pass http://mybackend;
    header_filter_by_lua 'ngx.header.Foo = "blah" ';
}
```

## 27. header\_filter\_by\_lua\_block

语法：

```
header_filter_by_lua_block { lua-script }
```

上下文：http、server、location、location if。

过程：output-header-filter。

说明：和 `header_filter_by_lua` 的区别仅是代码放在 `{ }` 内。例如：，

```
header_filter_by_lua_block {
    ngx.header["content-length"] = nil
}
```

## 28. header\_filter\_by\_lua\_file

语法：

```
header_filter_by_lua_file <path-to-lua-script-file>
```

上下文：http、server、location、location if。

阶段：output-header-filter。

说明：与 `header_filter_by_lua` 仅两点不同，一是代码放在文件中，二是从 v0.5.0rc32 版本开始，支持 Lua/LuaJIT 字节码。

## 29. body\_filter\_by\_lua

语法：

```
body_filter_by_lua <lua-script-str>
```

上下文：http、server、location、location if。

阶段：output-body-filter。

说明：从 v0.9.17 版本开始，推荐使用 `body_filter_by_lua_block` 代替。

使用 Lua 代码定义了一个应答包体过滤器。

输入的数据块通过 `ngx.arg[1]` 传递（作为 Lua 字符串值），“eof”标志指示应答包体数据流结束，标志存放在 `ngx.arg[2]` 中（作为 Lua 布尔类型值）。

在底层实现上，“eof”标志只是主请求 `last_buf` 或子请求 `last_in_chain` 链型缓冲区的标志。输出数据流可以被立即中断：

```
return ngx.ERROR
```

可以用于立即中断数据不正确、不完整以及无效的应答，以节省时间。

Lua 代码可以通过 `ngx.arg[1]` 传递编辑过的输入数据块到下游输出过滤器，数据作为一个 Lua 字符串或一个 Lua 字符串表。例如，在应答包体里转换所有的小写字符，可以这样写：

```
location / {
    proxy_pass http://mybackend;
    body_filter_by_lua 'ngx.arg[1] = string.upper(ngx.arg[1]);'
}
```

当向 `ngx.arg[1]` 传递一个 nil 值或空字符串时，不会有任何数据块向下游传递。同样地，可以通过在 `ngx.arg[2]` 中传递“eof”实现目的，例如：

```
location /t {
    echo hello world;
    echo hiya globe;

    body_filter_by_lua '
        local chunk = ngx.arg[1]
        if string.match(chunk, "hello") then
            ngx.arg[2] = true -- new eof
            return
        end

        -- just throw away any remaining chunk data
        ngx.arg[1] = nil
    '
}
```

输入 GET /t 将得到如下输出：

```
hello world
```

当包体过滤器看到一块包含着单词“hello”的数据块，将马上设置 eof 标志为真，结果是马上中断了后续处理，但本次应答仍然有效。

当 Lua 代码改变应答报文长度的时候，需要清除 Content-Length 应答头域，强迫流输出，例如：

```
location /foo {
    # fastcgi_pass/proxy_pass/...

    header_filter_by_lua_block { ngx.header.content-length = nil }
    body_filter_by_lua 'ngx.arg[1] = string.len(ngx.arg[1]) .. "\\n";'
}
```

注意, 下面的 API 因为系统限制, 不能在本阶段使用:

- 输出类 API, 如 ngx.say 和 ngx.send\_headers。
- 控制类 API, 如 ngx.redirect 和 ngx.exec。
- 子请求类 API, 如 ngx.location.capture 和 ngx.location.capture-multi。
- cosocket 类 API, 如 ngx.socket.tcp 和 ngx.req.socket。

一个输出过滤器在一个请求过程中可以被调用多次, 因为应答包体可能以块形式传输过来, 所以 Lua 代码也会被调用多次。

### 30. body\_filter\_by\_lua\_block

语法:

```
body_filter_by_lua_block { lua-script-str }
```

上下文: http、server、location、location if。

阶段: output-body-filter。

说明: 和 body\_filter\_by\_lua 不同的是代码放在 {} 中。例如:

```
body_filter_by_lua_block {
    local data, eof = ngx.arg[1], ngx.arg[2]
}
```

### 31. body\_filter\_by\_lua\_file

语法:

```
body_filter_by_lua_file <path-to-lua-script-file>
```

上下文: http、server、location、location if。

阶段: output-body-filter。

说明: 和 body\_filter\_by\_lua 有两点区别, 一是代码存放在文件中, 二是支持字节码。

当文件是类似于 foo/bar.lua 的形式时, 会被转换成绝对值。

### 32. log\_by\_lua

语法:

```
log_by_lua <lua-script-str>
```

上下文: http、server、location、location if。

阶段: log。

说明: 从 v0.9.17 版本开始, 鼓励使用 log\_by\_lua\_block 配置项代替。

在 log 请求处理阶段, 运行 <lua-script-str> 中的 Lua 代码。在访问 log 之前运行本段代码, 不替换原有 log 处理。

注意, 下列 API 在此过程中暂时是禁用的。

- 输出类 API, 如 ngx.say 和 ngx.send\_headers。

- 控制类 API, 如 ngx.redirect 和 ngx.exec。
- 子请求类 API, 如 ngx.location.capture 和 ngx.location.capture\_multi。
- cosocket 类 API, 如 ngx.socket.tcp 和 ngx.req.socket。

下面是一个从 \$upstream\_response\_time 中汇集并产生平均数据的例子:

```
lua_shared_dict log_dict 5M;

server {
    location / {
        proxy_pass http://mybackend;

        log_by_lua '
            local log_dict = ngx.shared.log_dict
            local upstream_time = tonumber(ngx.var.upstream_response_time)

            local sum = log_dict:get("upstream_time-sum") or 0
            sum = sum + upstream_time
            log_dict:set("upstream_time-sum", sum)

            local newval, err = log_dict:incr("upstream_time-nb", 1)
            if not newval and err == "not found" then
                log_dict:add("upstream_time-nb", 0)
                log_dict:incr("upstream_time-nb", 1)
            end
        ';
    }

    location = /status {
        content_by_lua_block {
            local log_dict = ngx.shared.log_dict
            local sum = log_dict:get("upstream_time-sum")
            local nb = log_dict:get("upstream_time-nb")

            if nb and sum then
                ngx.say("average upstream response time: ", sum / nb,
                    " (", nb, " reqs)")
            else
                ngx.say("no data yet")
            end
        }
    }
}
```

### 33. log\_by\_lua\_block

语法:

```
log_by_lua_block { lua-script }
```

上下文: http、server、location、location if。

阶段: log。

说明: 和 log\_by\_lua 的区别仅是代码放在 {} 内。例如:

```
log_by_lua_block {
    print("I need no extra escaping here, for example: \r\nblah")
}
```

### 34. log\_by\_lua\_file

语法:

```
log_by_lua_file <path-to-lua-script-file>
```

上下文: http、server、location、location if。

阶段: log。

说明: 与 log\_by\_lua 仅有两点不同, 一是代码放在 <path-to-lua-script-file> 指定的 lua 文件中, 二是从 v0.5.0rc32 版本开始, 支持 Lua/LuaJIT 字节码。

### 35. balancer\_by\_lua\_block

语法:

```
balancer_by_lua_block { lua-script }
```

上下文: upstream。

阶段: content。

说明: 这个配置项运行 Lua 代码作为上游服务器配置中服务器的负载均衡器。例如:

```
upstream foo {
    server 127.0.0.1;
    balancer_by_lua_block {
        -- 使用 Lua 在这里实现一个动态负载均衡器, 做一些有趣的事情。
    }
}

server {
    location / {
        proxy_pass http://foo;
    }
}
```

Lua 负载均衡器可以和任意存在的 Nginx 上游模块 (如 ngx\_proxy 和 ngx\_fastcgi) 协作。同时, Lua 负载均衡器可以和标准的上游连接池机制配合工作, 如标准的 keepalive 配置指令。只要确保一个 upstream{} 上下文中的 keepalive 配置项在 blancer\_by\_lua\_block 配置项之后。

Lua 负载均衡器可以完全忽略 upstream{} 中定义的服务器, 可以从 lua-resty-core 库的 ngx.balance 模块中选择完全独立的服务器列表。当 Nginx 上游机制尝试处理来自类似于 proxy\_next\_upstream 配置项的请求时, 管理器在一个请求中可以被调用超过一次。

Lua 代码运行的上下文不支持挂起操作, 所以禁用可以挂起的 Lua API (如 cosocket 和轻线程)。如果确实需要这样做, 那么把需要挂起的操作在一个早期的过程管理器里做好, 结果通过 ngx.ctx 表传进来。



### 36. balancer\_by\_lua\_file

语法:

```
balancer_by_lua_file <path-to-lua-script-file>
```

上下文: upstream。

阶段: content。

说明: 和 balancer\_by\_lua\_block 仅两点不同, 一是代码放在 <path-to-lua-script-file> 指定的文件中, 二是从 v0.5.0rc32 版本开始, 支持 Lua/LuaJIT 字节码。

当指定了类似于 foo/bar.lua 这样的文件名时, 会被 Nginx 启动时通过 -p PATH 选项传进来的 PATH 参数转换成绝对路径。

### 37. lua\_need\_request\_body

语法:

```
lua_need_request_body <on|off>
```

默认:

```
lua_need_request_body <off>
```

上下文: http、server、location、location if。

阶段: 依赖于用法。

说明: 决定请求的包体数据在 rewrite/access/access\_by\_lua\* 之前是否被读取。默认地, Nginx 核心不读取客户端的请求包体, 这个配置项可以打开或在 Lua 代码内通过调用 ngx.req.read\_body 函数读取。

通过 \$request\_body 读取请求包体数据, client\_body\_buffer\_size 值必须和 client\_max\_body\_size 一样, 因为如果包体内容长度超过 client\_body\_buffer\_size 但小于 clieng\_max\_body\_size, Nginx 就会把数据缓冲到硬盘上, 则 \$request\_body 会变空。如果当前 location 包含 rewrite\_by\_lua\* 配置项, 请求包包体会在 rewrite\_by\_lua\* 代码之前被类似于 rewrite 阶段读取, 同样地, 如果 content\_by\_lua 声明了, 请求包体则会在被 content 管理器处理过才会运行 (请求的包体会在 content 阶段被读取)。推荐使用 ngx.req.read\_body 和 ngx.req.discard\_body 函数, 实现精细的控制。

### 38. ssl\_certificate\_by\_lua\_block

语法:

```
ssl_certificate_by_lua_block { lua-script }
```

上下文: server。

阶段: right-before-SSL-handshake。

在 Nginx 和下游服务开始一个 SSL 握手操作时将运行本配置项的 Lua 代码。

这对于设置 SSL 证书链和选择一个适当的私有密钥是很重要的, 对于非阻塞地从远端

获取握手配置信息（使用 `cosocket` API）等也是很有用的。用纯 Lua 在这里做一些请求前的 OCSP stapling 交互是非常合适的。

另一个典型应用的是在 `lua-resty-limit-traffic` 库的帮助下非阻塞地进行 SSL 握手流量管理。还可以在客户端请求 SSL 握手时做一些有趣的事情，例如，使用 SSLv3 协议踢出旧的 SSL 客户端。Lua-resty-core 库提供的 `ngx.ssl` 和 `ngx.ocsp` 模块在这种场合下非常有用。可以使用 Lua API 在当前建立起的 SSL 连接上维护 SSL 证书链或私有密钥等工作。当 Nginx/OpenSSL 通过 SSL 会话 ID 或 TLS 会话票据成功恢复了 SSL 会话，或者这个管理器在 Nginx 初始化了一个完整的 SSL 握手后才会运行，其他情况下不会运行。

下面是一个使用 `ngx.ssl` 模块的例子：

```
server {
    listen 443 ssl;
    server_name test.com;

    ssl_certificate_by_lua_block {
        print("About to initiate a new SSL handshake!")
    }

    location / {
        root html;
    }
}
```

在当前的 SSL 会话中，Lua 代码不能立即捕获 Lua 异常，所以使用 `ngx.exit` 系统调用，检查 `ngx.ERROR` 一类的错误码。这个 Lua 运行上下文不支持挂起，所以禁止使用可以挂起的 Lua API（`cosocket`、`sleeping`、`协程`）。

注意，尽管使用者可能不使用静态证书和私有密钥，还是要配置 `ssl_certificate` 和 `ssl_certificate_key` 配置项，因为 Nginx 核心需要使用，否则将会看到下面的错误：

```
nginx: [emerg] no ssl configured for the server
```

这个配置项当前需要下面的 Nginx 核心补丁：

<http://mailman.nginx.org/pipermail/nginx-devel/2016-January/007748.html>

OpenResty 1.9.7.2 以上版本内的 Nginx 包已经打包了这个补丁。

另外，还需要至少 OpenSSL 1.0.2e 以支持本配置项。

### 39. `ssl_certificate_by_lua_file`

语法：

```
ssl_certificate_by_lua_file <path-to-lua-script-file>
```

配置项：server。

过程：right-before-SSL-handshake。

说明：和 `ssl_certificate_by_lua_block` 仅有两点不同，一是代码放在文件中，二是支持

字节码。文件名类似 foo/bar.lua 这种风格时，会被系统使用 server prefix 转换为绝对路径。

#### 40. ssl\_session\_fetch\_by\_lua\_block

语法：

```
ssl_session_fetch_by_lua_block { lua-script }
```

上下文：http。

过程：right-before-SSL-handshake。

说明：这个配置项根据下游服务请求的 SSL 握手会话 ID 载入 Lua 代码查询和载入 SSL 会话（如果需要）。Lua API 获取当前会话的 ID，并且通过 ngx.ssl.session（lua-resty-core 提供的）装载缓存的 SSL 会话数据。可以在这里使用需要挂起协程或阻塞的 API，如 ngx.sleep 和 cosocket。

这个管理器和 ssl\_session\_store\_by\_lua\* 钩子管理器，可以一起实现分布式缓存机制（依赖于 cosocket API 等）。如果缓存的 SSL 会话找到并且装入当前会话的上下文，SSL 会话可以立即重启，并且可以跨过 SSL 握手进程，大量节省 CPU 时间。

注意，TLS 会话的 tickets 是非常不同的，它的客户端在会话 tickets 使用时缓存 SSL 会话状态。SSL 会话重启基于 TLS 会话 tickets 自动处理，不依赖于这个钩子管理器。这个钩子管理器主要对旧的只能通过会话 ID 工作的 SSL 客户端有意义，服务于这种情况。

当同一时刻指定了 ssl\_certificate\_by\_lua\*，这个管理器通常在它们之前运行。当找到 SSL 会话并且成功载入当前 SSL 连接，SSL 会话会跳过 ssl\_certificate\_by\_lua 钩子管理器重启。这种情况下，Nginx 同样会跳过 ssl\_session\_store\_by\_lua\_block。当需要在一个主流浏览器简单测试这个钩子的时候，可以临时把下面这行加入到 https 上下文中，临时禁用 TLS 会话支持：

```
ssl_session_tickets off;
```

发布系统的时候不要忘记把这行去掉。如果要使用 OpenResty 1.11.2.1 或后面版本配套的官方预编译包，需要做好下面几件事情。

1) 如果使用的 OpenSSL 库不是 OpenResty 提供的，那么需要为 OpenSSL 1.0.2h 及以后版本打上下面的补丁程序：

```
https://github.com/openresty/openresty/blob/master/patches/openssl-1.0.2h-sess\_set\_get\_cb\_yield.patch
```

2) 如果没有使用 OpenResty 1.11.2.1 或后续版本中的 Nginx，那么需要在标准的 Nginx 1.11.2 或以后版本打上下面的补丁程序：

```
http://openresty.org/download/nginx-1.11.2-nonblocking\_ssl\_handshake\_hooks.patch
```

#### 41. ssl\_session\_fetch\_by\_lua\_file

语法：

```
ssl_session_fetch_by_lua_file <path-to-lua-script-file>
```

上下文: http。

阶段: right-before-SSL-handshake。

说明: 和 `ssl_session_fetch_by_lua_block` 仅有两点不同, 一是代码放在文件中, 二是支持字节码。当使用 `foo/bar.lua` 类的文件名时, 会被系统转换为绝对路径。

## 42. ssl\_session\_store\_by\_lua\_block

语法:

```
ssl_session_store_by_lua_block { lua-script }
```

上下文: http。

阶段: right-after-SSL-handshake。

说明: 这个配置项运行 Lua 代码并保存会话 ID 对应的 SSL 会话。保存和缓冲 SSL 会话数据, 用于后续 SSL 连接恢复 (重启), 目的是节省昂贵的 CPU 时间。禁止需要挂起协程类的 API, 如 `ngx.sleep` 和 `cosockets` 等。如果需要使用, 可使用 `ngx.timer.at` 创建一个零延迟的时钟异步地保存 SSL 会话数据到其他服务 (如 Redis 或 Memcached)。Lua-resty-core 库的 `ngx.ssl.session` 提供了获取当前会话 ID 以及和会话状态数据关系的 API。

当需要在一个流行的浏览器简单测试这个钩子的时候, 可以临时把下面这行加入 https 上下文中, 临时禁用 TLS 会话支持:

```
ssl_session_tickets off;
```

发布系统的时候不要忘记把这行去掉。

## 43. ssl\_session\_store\_by\_lua\_file

语法:

```
ssl_session_store_by_lua_file <path-to-lua-script-file>
```

上下文: http。

阶段: right-before-SSL-handshake。

说明: 与 `ssl_session_store_by_lua_block` 仅有两点不同, 一是代码放在文件中, 二是支持字节码。文件名同样会被 `server prefix` 转换为绝对路径。

## 44. lua\_shared\_dict

语法:

```
lua_shared_dict <name><size>
```

默认: no。

上下文: http。

阶段: 依赖于用法。

说明: 声明一个共享内存块 `<name>`, 使用 `ngx.shared.<name>` 路径提供一块用于存储

的内存。共享内存被用于向当前 Nginx 服务中所有的工作进程共享。<size> 参数接受的单位为 k 和 m。

```
http {
    lua_shared_dict dogs 10m;
    ...
}
```

#### 45. lua\_socket\_connect\_timeout

语法:

```
lua_socket_connect_timeout <time>
```

默认: lua\_socket\_connect\_timeout 60s。

上下文: http、server、location。

说明: 这个配置项控制 TCP/UNIX-domain 套接字的默认值, 这个值可以在程序中使用 `settimeout` 和 `settimeouts` 方法进行覆盖。<time> 参数可以是数值型, 单位为秒 (s)、毫秒 (ms)、分钟 (m)。默认单位是秒。默认值是 60 秒。

#### 46. lua\_socket\_send\_timeout

语法:

```
lua_socket_send_timeout <time>
```

默认:

```
lua_socket_send_timeout 60s
```

上下文: http、server、location。

说明: 控制 TCP/UNIX-domain 套接字的发送超时值, 可以被 `settimeout` 和 `settimeouts` 方法覆盖。<time> 参数可以是数值型, 单位为秒 (s)、毫秒 (ms)、分钟 (m)。默认单位是秒。默认值是 60 秒。

#### 47. lua\_socket\_send\_lowat

语法:

```
lua_socket_send_lowat <size>
```

默认:

```
lua_socket_send_lowat 0
```

上下文: http、server、location。

说明: 控制 cosocket 发送缓冲区的低水位线。

#### 48. lua\_socket\_read\_timeout

语法:

```
lua_socket_read_timeout <time>
```

默认:

```
lua_socket_read_timeout 60s
```

上下文: http、server、location。

过程: 依赖于用法。

说明: 设置 TCP/UNIX-domain 套接字默认接收超时值, 会影响 `receiveuntil` 方法。这个值可以被 `settimeout` 和 `settimeouts` 方法的值所覆盖。`<time>` 参数可以是数值型, 单位为秒 (s)、毫秒 (ms)、分钟 (m)。默认单位是秒。默认值是 60 秒。

#### 49. lua\_socket\_buffer\_size

语法:

```
lua_socket_buffer_size <size>
```

默认:

```
lua_socket_buffer_size 4k/8k
```

上下文: http、server、location。

说明: 声明 `cosocket` 读操作作用到的缓冲区大小。这个缓冲区不需要很大去缓冲所有的数据, 因为 `cosocket` 支持 100% 的非缓冲读取和解析。所以, 1B 的缓冲区同样可以正常工作, 只是性能会变得很差。

#### 50. lua\_socket\_pool\_size

语法:

```
lua_socket_pool_size <size>
```

默认:

```
lua_socket_pool_size 30
```

上下文: http、server、location。

说明: 声明每一个 `cosocket` 对应到远程服务的连接数量限制。这是一个连接池参数。默认每个连接池是 30 个连接。当连接池超过有效的限制时, 最后一个空闲池中的连接将被关闭, 释放空间给新的连接。

注意:

- 1) 连接池是针对工作进程的, 而不是针对服务器的。
- 2) 当连接超过最大连接池大小时, 会按照 LRU 算法回收空闲连接供新连接使用。
- 3) 连接池中的空闲连接出现异常时会自动被移除。
- 4) 连接池是通过 IP 和 port 标识的, 即相同的 IP 和 port 会使用同一个连接池 (即使是不同类型的客户端, 如 Redis、Memcached)。
- 5) 连接池第一次 `set_keepalive` 时, 连接池大小就确定下了, 不会再变更。

6) cosocket 的连接池是针对工程进程的，并不针对服务器，所以这个配置会应用到每一个工作进程上。

### 51. lua\_socket\_keepalive\_timeout

语法:

```
lua_socket_keepalive_timeout <time>
```

默认:

```
lua_socket_keepalive_timeout 60s
```

上下文: http、server、location。

说明: 设置连接在 cosocket 内建连接池中的最大空闲时间。当超时发生时, 空闲连接会被关闭并从池中删除, 这个设置可以被 setkeepalive 方法覆盖。<time> 参数可以是数值型, 单位为秒 (s)、毫秒 (ms)、分钟 (m)。默认单位是秒, 默认值是 60 秒。

### 52. lua\_socket\_log\_errors

语法:

```
lua_socket_log_errors on|off
```

默认:

```
lua_socket_log_errors on
```

上下文: http、server、location。

说明: 这个配置用于切换错误日志, 当 cosocket 的 TCP 或 UDP 发生错误时会记录日志。如果自己已经做了适当的错误处理或自己在代码里做了日志, 推荐关闭这个配置, 以保证自己程序中日志的刷写, 节省资源。

### 53. lua\_ssl\_ciphers

语法:

```
lua_ssl_ciphers <ciphers>
```

默认:

```
lua_ssl_ciphers DEFAULT
```

上下文: http、server、location。

说明: 声明 tcpsock:sslhandshake 用到的 SSL/TLS 密码。这个密码用的是 OpenSSL 库生成的不可读的格式。具体可参见 openssl ciphers 命令。

### 54. lua\_ssl\_crl

语法:

```
lua_ssl_crl <file>
```

默认: no。

上下文: http、server、location。

说明: 指定一个 tcpsock:sslhandshake 进行 SSL/TLS DRL 通信时要用到的 PEM 证书文件。

### 55. lua\_ssl\_protocols

语法:

```
lua_ssl_protocols [SSLv2] [SSLv3] [TLSv1] [TLSv1.1] [TLSv1.2]
```

默认:

```
lua_ssl_protocols SSLv3 TLSv1 TLSv1.1 TLSv1.2
```

上下文: http、server、location。

说明: 使能指定的 SSL 协议, 用于与 tcpsock:sslhandshake 方法进行 SSL/TLS 通信。

### 56. lua\_ssl\_trusted\_certificate

语法:

```
lua_ssl_trusted_certificate <file>
```

默认: no。

上下文: http、server、location。

说明: 指定 CA 证书的路径。PEM 格式的 CA 证书用于 tcpsock:sslhandshake 方法与 SSL/TLS 通信。

### 57. lua\_ssl\_verify\_depth

语法:

```
lua_ssl_verify_depth <number>
```

默认:

```
lua_ssl_verify_depth 1
```

上下文: http、server、location。

说明: 设置证书链的检验深度。

### 58. lua\_http10\_buffering

语法:

```
lua_http10_buffering on|off
```

默认:

```
lua_http10_buffering on
```

上下文: http、server、location、location if。

说明: 使能和关闭 HTTP 1.0 或更老协议应答包的自动缓冲。这个缓冲机制主要用于



HTTP 1.0 协议保活，应答包包头如果有一个合适的 Content-Length，就需要使用这个机制保活。

如果 Lua 代码在发送包头之前明确设置了一个应答头的 Content-Length 头域（或者明确通过 ngx.send\_headers 或隐含地通过第一个 ngx.say 或 ngx.print 调用实现），那么 HTTP 1.0 应答缓冲机制将被关闭，甚至是这个配置项被打开的情况下。在一个流式方法（如 ngx.flush）发送一个很大的应答数据时，这个配置项必须打开，以优化内存占用。这个配置项默认是打开的。

### 59. rewrite\_by\_lua\_no\_postpone

语法：

```
rewrite_by_lua_no_postpone on|off
```

默认：

```
rewrite_by_lua_no_postpone off
```

上下文：http。

说明：控制是否推迟 rewrite\_by\_lua\* 配置项在 rewrite 请求处理阶段之后运行。默认地，这个配置项是关闭的，Lua 代码推迟到 rewrite 阶段之后运行。

### 60. access\_by\_lua\_no\_postpone

语法：

```
access_by_lua_no_postpone on|off
```

默认：

```
access_by_lua_no_postpone off
```

上下文：http。

说明：控制 Lua 代码是否推迟到 access\_by\_lua\* 配置项 access 请求处理阶段之后运行。默认地，这个配置项是关闭的，Lua 代码推迟到 access 阶段后运行。

### 61. lua\_transform\_underscores\_in\_response\_headers

语法：

```
lua_transform_underscores_in_response_headers on|off
```

默认：

```
lua_transform_underscores_in_response_headers on
```

上下文：http、server、location、location if。

说明：控制是否在应答头名字中使用下划线（\_）。

### 62. lua\_check\_client\_abort

语法：

```
lua_check_client_abort on|off
```

默认:

```
lua_check_client_abort off
```

上下文: http、server、location、location if。

说明: 控制是否检查过早的连接中断。

当这个配置项打开时, ngx\_lua 模块将监视下游连接过早关闭情况。当这种情况发生时, 将会调用用户的 Lua 回调函数 (通过 ngx.on\_abort 注册的) 或当未注册回调函数时把当前请求管理器上的所有 Lua 协程关闭并清理掉。如果客户端在 Lua 代码通过 ngx.req.socket 读取请求完成之前关闭了连接, ngx\_lua 将永远不会清理所有的协程并调用用户回调函数。

这种情况下, ngx.req.socket 将在返回时返回错误信息, 第一个值应该是 nil。在 TCP 保活关闭的情况下, 依赖于客户端优雅地关闭连接 (发送一个 FIN 包或类似的操作)。在实时的 Web 应用下, 极力推荐在系统打开 TCP keepalive, 以便及时发现半打开的 TCP 连接。例如, 在 Linux 上, 可以在 nginx.conf 中配置标准的 listen 配置项:

```
listen 80 so_keepalive=2s:2s:8;
```

在 FreeBSD 系统上只需要打开系统级的配置即可:

```
# sysctl net.inet.tcp.keeptv1=2000
# sysctl net.inet.tcp.keeptv2=2000
```

### 63. lua\_max\_pending\_timers

语法:

```
lua_max_pending_timers <count>
```

默认:

```
lua_max_pending_timers 1024
```

上下文: http。

说明: 设置最大允许的待定时钟数量。待定时钟是仍然没有过时的时钟之一。当超过这些限制时, ngx.timer.at 调用会立即返回 nil, 并且返回错误字符串 "too many pending timers"。

### 64. lua\_max\_running\_timers

语法:

```
lua_max_running_timers <count>
```

默认:

```
lua_max_running_timers 256
```

上下文: http。

说明：设置最大允许的运行时钟数量。运行时钟是正在运行的用户回调函数，即用户注册的时钟。当超过限制时，Nginx 将停止回调最新的过期时钟，并且记录一个错误信息——“N lua\_max\_running\_timers are not enough”，N 就是这里配置的数值。

## 27.4 小结

本章详细介绍了 ngx\_lua\_module 模块，包括 ngx\_lua\_module 总体约束、使用机制，Lua 配置顺序，ngx-lua 模块的安装和编译，以及 ngx-lua 模块的配置指令。

ngx-lua 模块对于 Nginx 是一个扩展模块，所以配置和使用都是基于 nginx.conf 的。这些配置指令在内部都将配置到 ngx\_lua 模块中。

## ngx\_lua API 详解

Nginx API 包含方法、常量、状态码、变量等，服务于方法。各种 `*_by_lua`、`*_by_lua_block` 和 `*_by_lua_file` 指令是 `nginx.conf` 到 Lua API 的桥梁和网关。本章描述的 Nginx Lua API 只能在这些指令内编写和运行。

### 28.1 概述

API 通过 `ngx` 和 `ndk` 暴露给 Lua。这些包默认在 `ngx_lua` 全局上下文有效。这些包可以放到扩展 Lua 模块中，例如：

```
local say = ngx.say

local _M = {}

function _M.foo(a)
    say(a)
end

return _M
```

不允许使用 `package.seeall` 标志，是因为它有很多种可见或不可见的有害影响。也可以直接使用 `require` 扩展 Lua 模块。

```
local ngx = require "ngx"
local ndk = require "ndk"
```

只能通过 Lua API 调用用户模式的网络 I/O 操作，否则可能会阻塞 Nginx 消息循环，导致性能大幅下降。可以使用标准的 Lua I/O 库进行数量相对小的磁盘操作，因为会显著阻塞

Nginx 处理循环，所以需要避免大型的读写操作。为了得到最佳的性能，推荐将所有的网络和磁盘操作委派给 Nginx 的子请求（通过 ngx.location.capture 及相似的方法）。

为了方便查询，表 28-1 列出了所有的 API 与常量，后面将详细解释每一个 API。

表 28-1 Nginx API 与常量

名称	名称
ngx.arg	ngx.req.append_body
ngx.var.VARIABLE	ngx.req.finish_body
核心常量	ngx.req.socket
ngx.nuu	ngx.exec
HTTP 方法常量	ngx.redirect
HTTP 状态常量	ngx.send_headers
Nginx 日志级别常量	ngx.headers_sent
print	ngx.print
ngx.ctx	ngx.say
ngx.location.capture	ngx.log
ngx.location.capture_multi	ngx.flush
ngx.status	ngx.exit
ngx.header.HEADER	ngx.eof
ngx.resp.get_headers	ngx.sleep
ngx.req.is_internal	ngx.escape_uri
ngx.req.start_time	ngx.unescape_uri
ngx.req.http_version	ngx.encode_args
ngx.req.raw_header	ngx.decode_args
ngx.req.get_method	ngx.encode_base64
ngx.req.set_method	ngx.decode_base64
ngx.req.set_uri	ngx.crc32_short
ngx.req.set_uri_args	ngx.crc32_long
ngx.req.get_uri_args	ngx.hmac_shal
ngx.req.get_post_args	ngx.md5
ngx.req.get_headers	ngx.md5_bin
ngx.req.set_header	ngx.shal_bin
ngx.req.clear_header	ngx.quote_sql_str
ngx.req.read_body	ngx.today
ngx.req.discard_body	ngx.time
ngx.req.get_body_data	ngx.now
ngx.req.get_body_file	ngx.update_time
ngx.req.set_body_data	ngx.localtime
ngx.req.set_body_file	ngx.utctime
ngx.req.init_body	ngx.cookie_time

(续)

名称	名称
ngx.http_time	tcpsock:receiveuntil
ngx.parse_http_time	tcpsock:close
ngx.is_subrequest	tcpsock:settimeout
ngx.re.match	tcpsock:settimeouts
ngx.re.find	tcpsock:setoption
ngx.re.gmatch	tcpsock:setkeepalive
ngx.re.sub	tcpsock:getreusedtimes
ngx.re.gsub	ngx.socket.connect
ngx.shared.DICT	ngx.get_phase
ngx.shared.DICT.get	ngx.thread.spawn
ngx.shared.DICT.get_stale	ngx.thread.wait
ngx.shared.DICT.set	ngx.thread.kill
ngx.shared.DICT.safe_set	ngx.on_abort
ngx.shared.DICT.add	ngx.timer.at
ngx.shared.DICT.safe_add	ngx.timer.running_count
ngx.shared.DICT.replace	ngx.timer.pending_count
ngx.shared.DICT.delete	ngx.config.subsystem
ngx.shared.DICT.incr	ngx.config.debug
ngx.shared.DICT.lpush	ngx.config.prefix
ngx.shared.DICT.rpush	ngx.config.nginx_version
ngx.shared.DICT.lpop	ngx.config.nginx_configure
ngx.shared.DICT.rpop	ngx.config.ngx_lua_version
ngx.shared.DICT.llen	ngx.worker.exiting
ngx.shared.DICT.flush_all	ngx.worker.pid
ngx.shared.DICT.flush_expired	ngx.worker.count
ngx.shared.DICT.get_keys	ngx.worker.id
ngx.socket.udp	ngx.semaphore
udpsock:setpeername	ngx.balancer
udpsock:send	ngx.ssl
udpsock:receive	ngx.ocsp
udpsock:close	ndk.set_var.DIRECTIVE
udpsock:settimeout	coroutine.create
ngx.socket.stream	coroutine.resume
ngx.socket.tcp	coroutine.yield
tcpsock:connect	coroutine.wrap
tcpsock:sslhandshake	coroutine.running
tcpsock:send	coroutine.status
tcpsock:receive	

## 28.2 API 与常量

### 1. ngx.arg

语法:

```
val = ngx.arg[index]
```

上下文: set\_by\_lua\*, body\_filter\_by\_lua\*。

说明: 读取请求包的变量。在 set\_by\_lua\* 指令中使用的时候, 这个参数表是只读的:

```
value = ngx.arg[n]
```

例如:

```
location/foo {
    set$a32;
    set$b56;

    set_by_lua$sum 'return tonumber(ngx.arg[1]) + tonumber(ngx.arg[2])'
    $a$b;

    echo $sum;
}
```

输出的值是 88, 两个参数分别是 32 和 56, 只读。

当参数表在 body\_filter\_by\_lua\* 使用的时候, 第一个元素存放输入数据, 第二个参数存放布尔型的 “eof” 标志值, 标示整个输出数据的结束。

数据块和 “eof” 标志传递给下游 Nginx 输出过滤器, 也可以直接用参数表元素覆盖。当 ngx.arg[1] 被设置为 nil 或空字符串时, 将不会有数据传递给下游输出过滤器。

### 2. ngx.var.VARIABLE

语法:

```
ngx.var.VAR_NAME
```

上下文: set\_by\_lua\*、rewrite\_by\_lua、access\_by\_lua、content\_by\_lua、header\_filter\_by\_lua、body\_filter\_by\_lua、log\_by\_lua\*。

说明: 用于读写 Nginx 参数的值。例如:

```
value = ngx.var.some_nginx_variable_name
ngx.var.some_nginx_variable_name = value
```

注意, 只有预先定义的 Nginx 变量可以写入, 例如:

```
location/foo {
    set$my_var'';          # 提前创建 $my_var 变量
    content_by_lua_block {
        ngx.var.my_var = 123;
    }
    ...
}
```

```
}
```

确切地说，不能在运行中创建 Nginx 变量。一些指定的 Nginx 变量，如 \$args 和 \$limit\_rate 可以被赋值，大多数变量是不可以被赋值的，如 \$query\_string、\$arg\_PARAMETER、\$http\_NAME。

Nginx regex 正则 API 组通过 \$1、\$2、\$3 和其他方式获取变量，所以，ngx.var[1]、ngx.var[2]、ngx.var[3] 这种方式可以很好地工作。

设置 ngx.var.Foo 为 nil 值将取消 \$Foo 这个变量。

```
ngx.var.args=nil
```

注意，在读取变量的时候，Nginx 会在预请求内存池（只有在请求终止才会被释放）申请内存。所以，当需要重复读取 Nginx 变量的时候，需要把参数的值缓存到自己的 Lua 变量内。例如：

```
local val = ngx.var.some_var
--- 后面将重复使用变量
```

注意预防内存泄漏。另一个办法是使用 ngx.ctx 表缓存数据。未定义的 Nginx 变量被置为 nil，定义但未初始化的变量被置为空的字符串。这个 API (ngx.var.\*) 需要相对昂贵的元操作，应避免在频繁使用的代码段中使用。

ngx.var 的作用非常广范，例如：

- 在 ngx\_lua 中访问 Nginx 内置变量 ngx.var.arg\_PARAMETER 即可获得 GET 操作 PARAMETER 参数的内容。
- 访问 ngx.var.http\_HEADER 即可获得请求头 HEADER 的内容。对于常见的特殊头 (content-type、cookie 等)，Nginx 使用了特殊的变量来独立保存。例如，“content-type”头可以通过 ngx.var.content\_type 变量取得。
- 通过 ngx.var.request\_body 获得完整的 POST 请求体数据（注意，由于 Nginx 默认在处理请求前不自动读取请求包体，所以需要使用 ngx.req.read\_body 方法先读取包体数据或使用 lua\_need\_request\_body 配置指令，否则该变量内容始终为空）。

### 3. 核心常量

上下文：init\_by\_lua\*、set\_by\_lua\*、rewrite\_by\_lua\*、access\_by\_lua\*、content\_by\_lua\*、header\_filter\_by\_lua\*、body\_filter\_by\_lua\*、\*log\_by\_lua\*、ngx.timer.\*、balancer\_by\_lua\*、ssl\_certificate\_by\_lua\*、ssl\_session\_fetch\_by\_lua\*、ssl\_session\_store\_by\_lua\*。

例如：

```
ngx.OK (0)
ngx.ERROR (-1)
ngx.AGAIN (-2)
ngx.DONE (-4)
ngx.DECLINED (-5)
```



注意，只有 3 个常量被 Nginx Lua API 使用（ngx.exit 接受 NGX\_OK、NGX\_ERROR、NGX\_DECLINED 作为输入）。

#### 4. ngx.null

说明：ngx.null 常量是一个轻量级用户数据，通常用于在 Lua 表中表示 nil 值，和 lua-cjson 库中的 cjson.null 常量一样。

#### 5. HTTP 方法常量

上下文：init\_by\_lua\*、set\_by\_lua\*、rewrite\_by\_lua\*、access\_by\_lua\*、content\_by\_lua\*、header\_filter\_by\_lua\*、body\_filter\_by\_lua\*、log\_by\_lua\*、ngx.timer.\*、balancer\_by\_lua\*、ssl\_certificate\_by\_lua\*、ssl\_session\_fetch\_by\_lua\*、ssl\_session\_store\_by\_lua\*。

例如：

```
ngx.HTTP_GET
ngx.HTTP_HEAD
ngx.HTTP_PUT
ngx.HTTP_POST
ngx.HTTP_DELETE
ngx.HTTP_OPTIONS (added in the v0.5.0rc24 release)
ngx.HTTP_MKCOL (added in the v0.8.2 release)
ngx.HTTP_COPY (added in the v0.8.2 release)
ngx.HTTP_MOVE (added in the v0.8.2 release)
ngx.HTTP_PROPFIND (added in the v0.8.2 release)
ngx.HTTP_PROPPATCH (added in the v0.8.2 release)
ngx.HTTP_LOCK (added in the v0.8.2 release)
ngx.HTTP_UNLOCK (added in the v0.8.2 release)
ngx.HTTP_PATCH (added in the v0.8.2 release)
ngx.HTTP_TRACE (added in the v0.8.2 release)
```

这些常量经常在 ngx.location.capture 和 ngx.location.capture\_multi 调用中用到。

#### 6. HTTP 状态常量

上下文：init\_by\_lua\*、set\_by\_lua\*、rewrite\_by\_lua\*、access\_by\_lua\*、content\_by\_lua\*、header\_filter\_by\_lua\*、body\_filter\_by\_lua\*、log\_by\_lua\*、ngx.timer.\*、balancer\_by\_lua\*、ssl\_certificate\_by\_lua\*、ssl\_session\_fetch\_by\_lua\*、ssl\_session\_store\_by\_lua\*。

例如：

```
value = ngx.HTTP_CONTINUE (100) (first added in the v0.9.20 release)
value = ngx.HTTP_SWITCHING_PROTOCOLS (101) (first added in the v0.9.20 release)
value = ngx.HTTP_OK (200)
value = ngx.HTTP_CREATED (201)
value = ngx.HTTP_ACCEPTED (202) (first added in the v0.9.20 release)
value = ngx.HTTP_NO_CONTENT (204) (first added in the v0.9.20 release)
value = ngx.HTTP_PARTIAL_CONTENT (206) (first added in the v0.9.20 release)
value = ngx.HTTP_SPECIAL_RESPONSE (300)
value = ngx.HTTP_MOVED_PERMANENTLY (301)
value = ngx.HTTP_MOVED_TEMPORARILY (302)
value = ngx.HTTP_SEE_OTHER (303)
value = ngx.HTTP_NOT_MODIFIED (304)
```

```

value = ngx.HTTP_TEMPORARY_REDIRECT (307) (first added in the v0.9.20 release)
value = ngx.HTTP_BAD_REQUEST (400)
value = ngx.HTTP_UNAUTHORIZED (401)
value = ngx.HTTP_PAYMENT_REQUIRED (402) (first added in the v0.9.20 release)
value = ngx.HTTP_FORBIDDEN (403)
value = ngx.HTTP_NOT_FOUND (404)
value = ngx.HTTP_NOT_ALLOWED (405)
value = ngx.HTTP_NOT_ACCEPTABLE (406) (first added in the v0.9.20 release)
value = ngx.HTTP_REQUEST_TIMEOUT (408) (first added in the v0.9.20 release)
value = ngx.HTTP_CONFLICT (409) (first added in the v0.9.20 release)
value = ngx.HTTP_GONE (410)
value = ngx.HTTP_UPGRADE_REQUIRED (426) (first added in the v0.9.20 release)
value = ngx.HTTP_TOO_MANY_REQUESTS (429) (first added in the v0.9.20 release)
value = ngx.HTTP_CLOSE (444) (first added in the v0.9.20 release)
value = ngx.HTTP_ILLEGAL (451) (first added in the v0.9.20 release)
value = ngx.HTTP_INTERNAL_SERVER_ERROR (500)
value = ngx.HTTP_METHOD_NOT_IMPLEMENTED (501)
value = ngx.HTTP_BAD_GATEWAY (502) (first added in the v0.9.20 release)
value = ngx.HTTP_SERVICE_UNAVAILABLE (503)
value = ngx.HTTP_GATEWAY_TIMEOUT (504) (first added in the v0.3.1rc38 release)
value = ngx.HTTP_VERSION_NOT_SUPPORTED (505) (first added in the v0.9.20 release)
value = ngx.HTTP_INSUFFICIENT_STORAGE (507) (first added in the v0.9.20 release)

```

## 7. Nginx 日志级别常量

上下文: `init_by_lua*`、`init_worker_by_lua*`、`set_by_lua*`、`rewrite_by_lua*`、`access_by_lua*`、`content_by_lua*`、`header_filter_by_lua*`、`body_filter_by_lua*`、`log_by_lua*`、`ngx.timer.*`、`balancer_by_lua*`、`ssl_certificate_by_lua*`、`ssl_session_fetch_by_lua*`、`ssl_session_store_by_lua*`。

例如:

```

ngx.STDERR
ngx.EMERG
ngx.ALERT
ngx.CRIT
ngx.ERR
ngx.WARN
ngx.NOTICE
ngx.INFO
ngx.DEBUG

```

这些常量经常被 `ngx.log` 方法调用。

## 8. print

语法:

```
print(...)
```

上下文: `init_by_lua*`、`init_worker_by_lua*`、`set_by_lua*`、`rewrite_by_lua*`、`access_by_lua*`、`content_by_lua*`、`header_filter_by_lua*`、`body_filter_by_lua*`、`log_by_lua*`、`ngx.timer.*`、`balancer_by_lua*`、`ssl_certificate_by_lua*`、`ssl_session_fetch_by_lua*`、`ssl_session_store`

by\_lua\*。

说明：以 ngx.NOTICE 级别把参数的内容写到 error.log 文件，等同于

```
ngx.log(ngx.NOTICE, ...)
```

当 Lua 布尔型值放在一个字符串中，形式为 “true” 或 “false” 时，可以接受 nil 参数，并且结果也将放在一个字符串内，内容为 “nil”。Nginx 核心限制消息最长为 2048B。限制包括结尾的换行符和前导的时间戳。如果消息尺寸超过限制，Nginx 将截断消息。这个限制可以手工修改：修改 src/core/ngx\_log.h 中的 NGX\_MAX\_ERROR\_STR 宏定义。

## 9. ngx.ctx

上下文：init\_worker\_by\_lua\*、set\_by\_lua\*、rewrite\_by\_lua\*、access\_by\_lua\*、content\_by\_lua\*、header\_filter\_by\_lua\*、body\_filter\_by\_lua\*、log\_by\_lua\*、ngx.timer.\*、balancer\_by\_lua\*。

说明：这个表存放每个请求的 Lua 上下文数据，并且为当前请求保存一个生命周期（作为 Nginx 变量）。

考虑下面的例子：

```
location/test {
    rewrite_by_lua_block {
        ngx.ctx.foo = 76
    }
    access_by_lua_block {
        ngx.ctx.foo = ngx.ctx.foo + 3
    }
    content_by_lua_block {
        ngx.say(ngx.ctx.foo)
    }
}
```

GET /test 将输出

79

ngx.ctx.foo 变量经过了 across、rewrite 和 content 处理过程。每一个请求，包括子请求，都有自己对应的表副本，例如：

```
location/sub {
    content_by_lua_block {
        ngx.say("sub pre: ", ngx.ctx.blah)
        ngx.ctx.blah = 32
        ngx.say("sub post: ", ngx.ctx.blah)
    }
}

location /main {
    content_by_lua_block {
        ngx.ctx.blah = 73
        ngx.say("main pre: ", ngx.ctx.blah)
```

```

        local res = ngx.location.capture("/sub")
        ngx.print(res.body)
        ngx.say("main post: ", ngx.ctx.blah)
    }
}

```

调用 /main 将得到输出:

```

main pre: 73
sub pre: nil
sub post: 32
main post: 73

```

在子请求中编辑 ngx.ctx.blah 不影响父请求中的值。这是因为有两个分离的 ngx.ctx.blah。

内部重定向将销毁原请求的 ngx.ctx 数据，新的请求将拥有一个新的 ngx.ctx 表，例如:

```

location/new {
    content_by_lua_block {
        ngx.say(ngx.ctx.foo)
    }
}

location/orig {
    content_by_lua_block {
        ngx.ctx.foo = "hello"
        ngx.exec("/new")
    }
}

```

读取 /orig 将得到:

```

nil

```

任意的数据，包括 Lua 闭合函数或嵌套表，都可以插入这个表中，同时允许注册自定义元方法，并支持使用一个新的 Lua 表覆盖 ngx.ctx 表，例如:

```

ngx.ctx= { foo =32, bar =54 }

```

当在 init\_worker\_by\_lua\* 中使用的时候，这张表和处理器拥有相同的生命周期。ngx.ctx 查找需要调用相对昂贵的原函数，并且比自己通过函数参数传递数据速度慢，所以，考虑到系统整体性能，不要使用这个 API 保存自己函数参数。因为原方法的原因，不要在用户的 Lua 模块级别的 Lua 函数上下文范围外使用 ngx.ctx 表使用。例如，下面代码是很糟糕的:

```

-- mymodule.lua
local _M = {}

local ctx = ngx.ctx

function _M.main()
    ctx.foo="bar"
end

```

```
return _M
```

可使用下面的代码代替：

```
-- mymodule.lua
local _M = {}
```

```
function _M.main(ctx)
    ctx.foo="bar"
end
```

```
return _M
```

调用者通过函数参数显式传输 ctx 表。

## 10. ngx.location.capture

语法：

```
res = ngx.location.capture(uri, options?)
```

上下文：rewrite\_by\_lua\*、access\_by\_lua\*、content\_by\_lua\*。

说明：使用 URI 发起一个并发但是非阻塞的 Nginx 子请求。

Nginx 子请求提供了一个发起非阻塞内部请求有力的方法，以使用其他的功能性 location 的服务，如其他的 Nginx C 模块、ngx\_proxy、ngx\_fastcgi、ngx\_memc、ngx\_postgres、ngx\_drizzle 等。子请求都是模拟的 HTTP 接口，但并不支持扩展的 HTTP/TCP 通信。所有内部环节都是工作在 C 语言的级别。子请求和 HTTP 301/302（通过 ngx.redirect）重定向及内部重定向（通过 ngx.exec）完全不同。

通常，capture 初始化一个子请求之前需要读取请求包体（通过调用 ngx.req.read\_body 或配置 lua\_need\_request\_body 为 on）。这个 API 函数（同样有效于 ngx.location.capture\_multi）总是在内存中缓冲整个子请求应答包体。这样，如果需要处理一个很大的子请求应答，可以使用 cosocket 和流进行处理。

下面是一个基本的例子：

```
res = ngx.location.capture(uri)
```

返回一个有 4 个元素的 Lua 表：res.status、res.header、res.body、res.truncated。

- res.status 存放子请求应答的状态码。
- res.header 存储所有子请求应答报文头域，是一个 Lua 表。在多个应答值的情况下，值是 Lua 数组，按顺序存放所有的值。例如，如果子请求应答头包含下列的行：

```
Set-Cookie: a=3
Set-Cookie: foo=bar
Set-Cookie: baz=blah
```

res.header["Set-Cookie"] 的值被合并成一个表，值为 {"a=3", "foo=bar", "baz=blah"}。

- res.body 存储子请求应答的包体数据，数据可能会被截断。

- `res.truncated` 值是布尔型, `true` 表示 `res.body` 包含被截断的数据。数据截断只会发生在无法处理的情况下, 如远程突然中断了连接或传输过程中远程挂机等情况。

URI 参数可以被 URI 自己级联, 例如:

```
res = ngx.location.capture('/foo/bar?a=3&b=4')
```

@foo 这样的命名被 Nginx 核心限制在 URI 使用, 这种叫 location “命名 location”, 专供内部重定向使用, 一般用于 `error_page`、`try_files`。如果要创建仅供内部使用的 location, 则在配置 location 时加上 `internal` 指令。

options 是可选参数, 标示可选选项, 支持选项如下:

- `method` 指定子请求的方法, 只接受如 `ngx.HTTP_POST` 的常量。
- `body` 指定子请求的包体 (只接受字符串值)。
- `args` 指定子请求 URL 查询参数 (只接受字符串和 Lua 表)。
- `ctx` 指定一个 Lua 表代替子请求的 `ngx.ctx` 表, 可以是当前请求的 `ngx.ctx` 表, 使父请求和子请求共享相同的上下文表。
- `vars` 使用一个带值的 Lua 表设置子请求 Nginx 变量的值。
- `copy_all_vars` 声明是否复制当前请求 Nginx 变量到子请求。子请求对变量的编辑不会影响父请求。
- `share_all_vars` 声明是否将子请求的 Nginx 变量共享给父请求, 父请求对变量的修改不会影响子请求。这种方式用于难于调试的情形下对代码进行调试, 不能作为常规应用, 因为性能非常糟糕。只有在非常清楚自己在做什么的情况下才能打开这个选项。
- `always_forward_body` 选项设置为 `true`, 而 `body` 选项没有声明的时候, 当前请求的包体将传给子请求。通过 `ngx.req.read_body()` 或 `lua_need_request_body on` 读取的包体会直接交给子请求 (因为请求包体数据被缓存在内存中或临时文件中, 所以不用担心性能问题)。默认地, 当没有指定 `body` 选项时, 这个选项是 `false`, 当前请求的包体只会在子请求是 PUT 或 POST 方法时才会被转发。

发起一个 POST 子请求, 可以这样做:

```
res = ngx.location.capture(
    '/foo/bar',
    { method = ngx.HTTP_POST, body = 'hello, world' }
)
```

调用使用 POST 方法, `ngx.HTTP_GET` 是默认的方法。

`args` 选项可以通过扩展参数实现, 例如:

```
ngx.location.capture('/foo?a=1',
    { args = { b = 3, c = ':' } }
)
```

等同于

```
ngx.location.capture('/foo?a=1&b=3&c=%3a')
```

用到的参数键和值，将会按照 URI 规则组成一个完整的查询字符串。作为 args 参数传输的 Lua 表的格式和 ngx.encode\_args 的格式是一样的。

args 选项也可以使用普通的查询字符串：

```
ngx.location.capture('/foo?a=1',
    { args = 'b=3&c=%3a' } }
)
```

跟上面例子的结果是一样的。

share\_all\_vars 选项控制是否在当前请求和子请求间共享 Nginx 变量。如果该选项设置为 true，当前请求和相关的子请求将共享 Nginx 变量上下文。子请求修改变量将会影响当前请求。谨慎使用这个配置，避免变量上下文共享引起未知的上下文影响。

args、vars、copy\_all\_vars 选项通常有更好的替代方法。

默认地，这个选项是关闭的：

```
location/other {
    set$dog"$dog world";
    echo"$uri dog: $dog";
}

location/lua {
    set$dog'hello';
    content_by_lua_block {
        res = ngx.location.capture("/other",
            { share_all_vars = true });

        ngx.print(res.body)
        ngx.say(ngx.var.uri, ": ", ngx.var.dog)
    }
}
```

访问 location /lua 会得到输出：

```
/other dog: hello world
/lua: hello world
```

copy\_all\_vars 选项将父请求选项的变量复制了一份给子请求。改变子请求中的变量不会影响父请求中的变量。

```
location/other {
    set$dog"$dog world";
    echo"$uri dog: $dog";
}

location/lua {
    set$dog'hello';
    content_by_lua_block {
        res = ngx.location.capture("/other",
            { copy_all_vars = true });

        ngx.print(res.body)
```

```

        ngx.say(ngx.var.uri, ": ", ngx.var.dog)
    }
}

```

请求 GET /lua 将得到输出:

```

/other dog: hello world
/lua: hello

```

注意, 如果 `share_all_vars` 和 `copy_all_vars` 都设置为 `true`, 则 `share_all_vars` 优先。

除了上面的方法外, 可以通过 `vars` 选项向子请求传递变量值。这些变量将在共享或复制之后设置, 并且提供一个非常有效率的方法为 URL 参数编码并且进行 Base64 解码后传递给子请求。

```

location/other {
    content_by_lua_block {
        ngx.say("dog = ", ngx.var.dog)
        ngx.say("cat = ", ngx.var.cat)
    }
}

location /lua {
    set $dog '';
    set $cat '';
    content_by_lua_block {
        res = ngx.location.capture("/other",
            { vars = { dog = "hello", cat = 32 } });

        ngx.print(res.body)
    }
}

```

访问 /lua 会得到输出:

```

dog = hello
cat = 32

```

`ctx` 选项用于指定一个 Lua 表代替子请求的 `ngx.ctx`。

```

location/sub {
    content_by_lua_block {
        ngx.ctx.foo = "bar";
    }
}

location/lua {
    content_by_lua_block {
        local ctx = {}
        res = ngx.location.capture("/sub", { ctx = ctx })

        ngx.say(ctx.foo);
        ngx.say(ngx.ctx.foo);
    }
}

```



访问 GET /lua:

```
bar
nil
```

也可以在当前请求和子请求间共享相同的表:

```
location/sub {
    content_by_lua_block {
        ngx.ctx.foo = "bar";
    }
}
location/lua {
    content_by_lua_block {
        res = ngx.location.capture("/sub", { ctx = ngx.ctx })
        ngx.say(ngx.ctx.foo);
    }
}
```

通过 GET 访问 /lua 获得输出:

```
bar
```

`ngx.location.capture` 创建的子请求默认继承了当前请求所有的头域, 这可能会引发子请求应答中非预期中的影响。例如, 当我们使用标准的 `ngx_proxy` 模块服务于子请求时, 主请求的 “Accept-Encoding: gzip” 头域将引发结果被压缩, Lua 代码不能处理。原请求头应该忽略掉。在子请求的 `location` 配置中设置 `proxy_pass_request_headers` 为 `off`。

`body` 选项没指定, 并且 `always_forward_body` 选项为 `false` (默认值), `POST` 和 `PUT` 的子请求将从父请求继承请求包包体。

每个主请求对应的子请求例程有一个硬性限制, 老版本的 Nginx 限制 50 个子请求例程, 最近的版本, 从 Nginx 1.1.x 开始, 这个值提高到 200 个协程。当达到这个限制后, 将会在 `error.log` 里收到错误信息:

```
[error] 13983#0: *1 subrequests cycle while processing "/uri"
```

这个限制可以手工修改: 修改 `nginx/src/http/ngx_http_request.h` 中的 `NGX_HTTP_MAX_SUBREQUESTS` 宏。

## 11. ngx.location.capture\_multi

语法:

```
res1, res2, ... = ngx.location.capture_multi({ {uri, options?}, {uri, options?}, ... })
```

上下文: `rewrite_by_lua*`、`access_by_lua*`、`content_by_lua*`。

说明: 与 `ngx.location.capture` 很相似, 只是支持并发多个子连接请求。这个函数发起几个并行的子请求, 同时按请求顺序返回结果, 例如:

```
res1,res2,res3 = ngx.location.capture_multi{
```

```

    { "/foo", { args = "a=3&b=4" } },
    { "/bar" },
    { "/baz", { method = ngx.HTTP_POST, body = "hello" } },
}

if res1.status== ngx.HTTP_OKthen
...
end

if res2.body=="BLAH"then
...
end

```

这个函数在所有子请求中断之前不会返回，花费时间是最长请求的时间，比分别执行的时间总和要好很多。

当并发请求的数量不能确定的时候，可以在请求和应答中使用 Lua 表。

```

-- 构造请求表
local reqs = {}
table.insert(reqs, { "/mysql" })
table.insert(reqs, { "/postgres" })
table.insert(reqs, { "/redis" })
table.insert(reqs, { "/memcached" })

-- 发送请求并且等待返回
local resps = { ngx.location.capture_multi(reqs) }

-- 循环处理应答表
for i, resp in ipairs(resps) do
-- process the response table "resp"
end

```

ngx.location.capture 只是 ngx.location.capture\_multi 函数的一个指定格式，逻辑上来讲，ngx.location.capture 可以这样实现：

```

ngx.location.capture=
function (uri, args)
    return ngx.location.capture_multi({ {uri, args} })
end

```

## 12. ngx.status

上下文: set\_by\_lua\*、rewrite\_by\_lua\*、access\_by\_lua\*、content\_by\_lua\*、header\_filter\_by\_lua\*、body\_filter\_by\_lua\*、log\_by\_lua\*。

说明：当前请求的读写状态，应当在发送应答头之前调用和检查。

```

ngx.status= ngx.HTTP_CREATED
status = ngx.status

```

在应答包发送之后设置 ngx.status 不会产生影响，但是会在 error.log 里留下一行错误信息。

```
attempt to set ngx.status after sending out response headers
```

### 13. ngx.header.HEADER

语法:

```
ngx.header.HEADER = VALUE
value = ngx.header.HEADER
```

上下文: `rewrite_by_lua*`、`access_by_lua*`、`content_by_lua*`、`header_filter_by_lua*`、`body_filter_by_lua*`、`log_by_lua*`。

说明: 设置、添加或清除当前请求应答头。头域名称中的下划线 ( `_` ) 默认会被连字符替换 ( `-` )。这个转换可以通过将 `lua_transform_underscores_in_response_headers` 指令置为 `off` 关闭。头域名字是大小写敏感的。

下面是 3 个效果相等的操作:

```
ngx.header["Content-Type"] = 'text/plain'
ngx.header.content_type='text/plain';
ngx.header["X-My-Header"] = 'blah blah';
```

多值的头域可以这样设置:

```
ngx.header['Set-Cookie'] = {'a=32; path=/', 'b=4; path=/'}
```

将在应答头中得到结果:

```
Set-Cookie: a=32; path=/
Set-Cookie: b=4; path=/'
```

头域只接受 Lua 表 (因为单个头域只接受一个值, 所以只有表中最近的值才有效)。

```
ngx.header.content_type= {'a', 'b'}
```

等同于

```
ngx.header.content_type='b'
```

把一个头域设置为 `nil` 就等于从应答头域中移除了该头域:

```
ngx.header["X-My-Header"] = nil;
```

绑定一个空表, 起到的是相同的作用:

```
ngx.header["X-My-Header"] = {};
```

在应答头已经发送完成后设置 `ngx.header.HEADER` (或明确使用 `ngx.send_headers` 或明确使用 `ngx.print`) 将抛出一个 Lua 异常。读取 `ngx.header.HEADER` 将返回 `HEADER` 头域的值。头域中的下划线 ( `_` ) 同样会被连字符 ( `-` ) 替换, 并且头域是大小写敏感的。如果应答头没有完全准备好, 将会返回 `nil` 值, 这在 `header_filter_by_lua*` 上下文中特别有用, 例如:

```
location/test {
    set$footer'';

    proxy_pass http://some-backend;
```

```
header_filter_by_lua_block {
    if ngx.header["X-My-Header"] == "blah" then
        ngx.var.footer = "some value"
    end
}
```

```
echo_after_body$footer;
}
```

在多返回值的头域，所有的值都收集在一个 Lua 表中，例如，应答头：

```
Foo: bar
Foo: baz
```

将存放在

```
{"bar", "baz"}
```

读取 `ngx.header.Foo` 时将得到这个表。`ngx.header` 不是一个普通的 Lua 表，它不能通过 Lua `ipairs` 函数遍历。要读取请求头，可使用 `ngx.req.get_headers` 函数代替。

#### 14. ngx.resp.get\_headers

语法：

```
headers = ngx.resp.get_headers(max_headers?, raw?)
```

上下文：set\_by\_lua\*、rewrite\_by\_lua\*、access\_by\_lua\*、content\_by\_lua\*、header\_filter\_by\_lua\*、body\_filter\_by\_lua\*、log\_by\_lua\*、balancer\_by\_lua\*。

说明：返回一个容纳应答头域的 Lua 表。

```
local h = ngx.resp.get_headers()
for k, v in pairs(h) do
    ...
end
```

这个函数和 `ngx.req.get_headers` 相同，除了它是读取应答头。

#### 15. ngx.req.is\_internal

语法：

```
is_internal = ngx.req.is_internal()
```

上下文：set\_by\_lua\*、rewrite\_by\_lua\*、access\_by\_lua\*、content\_by\_lua\*、header\_filter\_by\_lua\*、body\_filter\_by\_lua\*、log\_by\_lua\*。

说明：返回一个布尔值，表明当前请求是否是一个内部请求。内部请求是 Nginx 服务器内部初始化代替客户端请求的一种请求。内部重定向的子请求全部都是内部请求。

#### 16. ngx.req.start\_time

语法：

```
secs = ngx.req.start_time()
```

上下文: `set_by_lua*`、`rewrite_by_lua*`、`access_by_lua*`、`content_by_lua*`、`header_filter_by_lua*`、`body_filter_by_lua*`、`log_by_lua*`。

说明: 返回一个描述请求创建时间的浮点数, 存放时戳 (包含毫秒)。

下面例子使用纯 Lua 模拟 `$request_time` 变量值:

```
local request_time = ngx.now() - ngx.req.start_time()
```

### 17. ngx.req.http\_version

语法:

```
num = ngx.req.http_version()
```

上下文: `set_by_lua*`、`rewrite_by_lua*`、`access_by_lua*`、`content_by_lua*`、`header_filter_by_lua*`。

说明: 以 Lua 数值型返回当前请求的 HTTP 协议版本。当前可能的值是 2.0、1.0、1.1、0.9。nil 表示是不认识的协议。

### 18. ngx.req.raw\_header

语法:

```
str = ngx.req.raw_header(no_request_line?)
```

上下文: `set_by_lua*`、`rewrite_by_lua*`、`access_by_lua*`、`content_by_lua*`、`header_filter_by_lua*`。

说明: 返回 Nginx 接收到的原始 HTTP 协议头。默认地, 请求中的换行符和回车符都包含在内。例如:

```
ngx.print(ngx.req.raw_header())
```

会输出下面类似的信息:

```
GET /t HTTP/1.1
Host: localhost
Connection: close
Foo: bar
```

可以指定可选的 `no_request_line` 参数以 true 值传入, 排除请求行, 例如:

```
ngx.print(ngx.req.raw_header(true))
```

输出将会是这样的:

```
Host: localhost
Connection: close
Foo: bar
```

这个方法仍然不能在 HTTP/2 下面工作。

### 19. ngx.req.get\_method

语法:

```
method_name = ngx.req.get_method()
```

上下文: set\_by\_lua\*、rewrite\_by\_lua\*、access\_by\_lua\*、content\_by\_lua\*、header\_filter\_by\_lua\*、balancer\_by\_lua\*。

说明: 获取当前请求中请求方法名。值是“GET”和“POST”这样的字符串,不是数值方法常量。如果当前请求是 Nginx 子请求,那么将会返回子请求的方法名字。

## 20. ngx.req.set\_method

语法:

```
ngx.req.set_method(method_id)
```

上下文: set\_by\_lua\*、rewrite\_by\_lua\*、access\_by\_lua\*、content\_by\_lua\*、header\_filter\_by\_lua\*。

说明: 使用 method\_id 参数覆盖当前请求的方法。目前只支持方法常量,如 ngx.HTTP\_POST 和 ngx.HTTP\_GET。如果当前是子请求,则会覆盖子请求的方法。

细节参见 ngx.req.get\_method 方法。

## 21. ngx.req.set\_uri

语法:

```
ngx.req.set_uri(uri, jump?)
```

上下文: set\_by\_lua\*、rewrite\_by\_lua\*、access\_by\_lua\*、content\_by\_lua\*、header\_filter\_by\_lua\*、body\_filter\_by\_lua\*。

说明: 使用 uri 重写当前请求的 URI。uri 参数必须是 Lua 字符串,不能是零长度,否则将会抛出一个 Lua 异常。可选的参数 jump 是布尔值,可以触发 location 的 ngx\_http\_rewrite\_module 模块的重写 (rewrite) 或跳转。当 jump 是 true (默认是 false) 时,这个函数将永远不会返回,它将告诉 Nginx 去重新匹配新的 location。除非当前请求的 URI 被修改过了,否则永远不会触发 location 跳转,这是默认的行为。当 jump 参数是 false 或不填,这个函数会返回,但没有返回值。jump 为 true,等价于 rewrite...last。

例如,下面的 Nginx 配置文件,

```
rewrite^ /foo last;
```

可以这样编码:

```
ngx.req.set_uri("/foo", true)
```

相似地,nginx.conf 如下:

```
rewrite^ /foo break;
```

不可以这样编码:

```
ngx.req.set_uri("/foo", false)
```

或者

```
ngx.req.set_uri("/foo")
```

在 `rewrite_by_lua*` 中, `jump` 只能设置为 `true`。在其他的上下文中, `jump` 是禁止的, 否则会抛出 Lua 异常。

一个更复杂的例子是调用正则表达式:

```
location/test {
    rewrite_by_lua_block {
        local uri = ngx.re.sub(ngx.var.uri, "^/test/(.*)", "/$1", "o")
        ngx.req.set_uri(uri)
    }
    proxy_pass http://my_backend;
}
```

和下面的代码等同:

```
location/test {
    rewrite^/test/(.*) /$1 break;
    proxy_pass http://my_backend;
}
```

注意, 不可能使用这个接口重写 URI 参数, 使用 `ngx.req.set_uri_args` 替换, 例如, Nginx config:

```
rewrite^ /foo?a=3? last;
```

可以编码为

```
ngx.req.set_uri_args("a=3")
ngx.req.set_uri("/foo", true)
```

或

```
ngx.req.set_uri_args({a = 3})
ngx.req.set_uri("/foo", true)
```

## 22. ngx.req.set\_uri\_args

语法:

```
ngx.req.set_uri_args(args)
```

上下文: `set_by_lua*`、`rewrite_by_lua*`、`access_by_lua*`、`content_by_lua*`、`header_filter_by_lua*`、`body_filter_by_lua*`。

说明: 使用 `args` 参数重写当前请求的 URI 查询参数。 `args` 参数可以是一个 Lua 字符串, 类似于:

```
ngx.req.set_uri_args("a=3&b=hello%20world")
```

或者一个以 `key/value` 形式存储参数对的 Lua 表, 类似于:

```
ngx.req.set_uri_args({ a = 3, b = "hello world" })
```

当使用 Lua 表的方法时，方法将会按 URL 的格式将键和值组成查询串。也支持多值参数：

```
ngx.req.set_uri_args({ a = 3, b = {5, 6} })
```

将编码成一个字符串：a=3&b=5&b=6。

## 23. ngx.req.get\_uri\_args

语法：

```
args = ngx.req.get_uri_args(max_args?)
```

上下文：set\_by\_lua\*、rewrite\_by\_lua\*、access\_by\_lua\*、content\_by\_lua\*、header\_filter\_by\_lua\*、body\_filter\_by\_lua\*、log\_by\_lua\*、balancer\_by\_lua\*。

说明：返回一个包含当前请求所有 URL 查询参数的 Lua 表。

```
location= /test {
    content_by_lua_block {
        local args = ngx.req.get_uri_args()
        for key, val in pairs(args) do
            if type(val) == "table" then
                ngx.say(key, ":", table.concat(val, ","))
            else
                ngx.say(key, ":", val)
            end
        end
    }
}
```

GET /test?foo=bar&bar=baz&bar=blah 将收到下面的应答包体：

```
foo: bar
bar: baz, blah
```

一个键的多个值按顺序地存放在表中。键和值按 URI 编码规则解码后生成。上面例子中的 URL：GET /test?a%20b=1%61+2 将得到：

```
a b: 1a 2
```

没有值的参数会被处理成布尔型参数。GET /test?foo&bar 会得到：

```
foo: true
bar: true
```

但是，在参数值是空字符串的情况下，情况又不一样。GET /test?foo=&bar= 将得到：

```
foo:
bar:
```

禁止空参数名这种情况。GET /test?=hello&=world 将得到一个空的输出。支持通过 \$args 实时更新查询参数：

```
ngx.var.args="a=3&b=42"
local args = ngx.req.get_uri_args()
```

args 表是这样的：



```
{a = 3, b = 42}
```

注意，默认支持最多 100 个请求参数（包括相同名字的参数），并且超过的额外参数被自动丢弃，以防止服务器攻击。max\_args 函数参数可以用来覆盖这个限制：

```
local args = ngx.req.get_uri_args(10)
```

这个参数可以设置成 0 移除限制，以处理所有的参数：

```
local args = ngx.req.get_uri_args(0)
```

## 24. ngx.req.get\_post\_args

语法：

```
args, err = ngx.req.get_post_args(max_args?)
```

上下文: rewrite\_by\_lua\*、access\_by\_lua\*、content\_by\_lua\*、header\_filter\_by\_lua\*、body\_filter\_by\_lua\*、log\_by\_lua\*。

说明：返回一个 Lua 表，容纳所有当前请求的 POST 查询参数（MIME 类型 application/x-www-form-urlencoded）。调用 ngx.req.read\_body 首先读取请求包体或打开 lua\_need\_request\_body 指令避免错误。

```
location= /test {
  content_by_lua_block {
    ngx.req.read_body()
    local args, err = ngx.req.get_post_args()
    if not args then
      ngx.say("failed to get post args: ", err)
      return
    end
    for key, val in pairs(args) do
      if type(val) == "table" then
        ngx.say(key, ": ", table.concat(val, ", "))
      else
        ngx.say(key, ": ", val)
      end
    end
  }
}
```

测试：

```
# Post request with the body 'foo=bar&bar=baz&bar=blah'
$ curl --data 'foo=bar&bar=baz&bar=blah' localhost/test
```

将得到类似的应答包体：

```
foo: bar
bar: baz, blah
```

一个参数有多个返回值，使用一个 Lua 表依次存放。键和值会使用 URI 编码规则正反向处理。按照上面的配置：

```
# POST request with body 'a%20b=1%61+2'
$ curl -d 'a%20b=1%61+2' localhost/test
```

将得到:

```
a b: 1a 2
```

没有 `=<value>` 部分的参数被当成布尔型对待。使用 “foo&bar” 作为包体 POST /test 会得到:

```
foo: true
bar: true
```

如果只是空字符串, 则结果不同。以 “foo=&bar=” 为请求包体 POST /test 将得到

```
foo:
bar:
```

禁止空参数名情况。“=hello&=world” 这个参数 POST 将会得到空输出。默认支持最多 100 个请求参数 (包括相同名字的参数), 并且将自动丢弃超过的额外参数, 以防止服务器攻击。

`max_args` 函数参数可以用来覆盖这个限制:

```
local args = ngx.req.get_uri_args(10)
```

这个参数可以设置成 0 移除限制, 以处理所有的参数:

```
local args = ngx.req.get_uri_args(0)
```

## 25. ngx.req.get\_headers

语法:

```
headers = ngx.req.get_headers(max_headers?, raw?)
```

上下文: `set_by_lua*`、`rewrite_by_lua*`、`access_by_lua*`、`content_by_lua*`、`header_filter_by_lua*`、`body_filter_by_lua*`、`log_by_lua*`。

说明: 返回一个 Lua 表容纳当前请求所有头域。

```
local h = ngx.req.get_headers()
for k, v in pairs(h) do
...
end
```

读取某一个头域:

```
ngx.say("Host: ", ngx.req.get_headers()["Host"])
```

注意, `ngx.var.HEADERAPI` 使用了核心的 `$http_HEADER` 变量, 所以更适用于读取单个头域。对于常见的特殊头 (`Content-Type`、`cookie` 等), NginX 使用了特殊的变量独立保存, 例如, “Content-Type” 头域可以通过 `ngx.var.content_type` 变量取得。

一个多实例的请求头域例子:

```

Foo: foo
Foo: bar
Foo: baz

```

`ngx.req.get_headers()` [“Foo”] 值是一个 Lua 表或数组，类似于：

```
{ "foo", "bar", "baz" }
```

注意，默认支持最多 100 个请求参数（包括相同名字的参数），并且超过的额外参数被自动丢弃，以防止服务器攻击。

`max_args` 函数参数可以用来覆盖这个限制：

```
local args = ngx.req.get_uri_args(10)
```

这个参数可以设置成 0 移除限制，以处理所有的参数：

```
local args = ngx.req.get_uri_args(0)
```

从 0.6.9 版本开始，Lua 表中的头域默认全部被转换成小写，除非 `raw` 参数设置为 `true`（默认为 `false`）。默认地，结果 Lua 表也增加了 `_index` 元方法，以小写形式索引头域。例如，一个请求头域有一个 `My-Foo-Header` 头域，下面的方法都可以取出对应的值：

```

ngx.say(headers.my_foo_header)
ngx.say(headers["My-Foo-Header"])
ngx.say(headers["my-foo-header"])

```

当 `raw` 参数设置为 `true` 时，将不能使用下标元方法。

## 26. ngx.req.set\_header

语法：

```
ngx.req.set_header(header_name, header_value)
```

上下文：`set_by_lua*`、`rewrite_by_lua*`、`access_by_lua*`、`content_by_lua*`、`header_filter_by_lua*`、`body_filter_by_lua*`。

说明：将当前请求名字为 `header_name` 的头域设置成 `header_value` 值，覆盖原有数据。默认地，所有通过初始化的 `ngx.location.capture` 和 `ngx.location.capture_multi` 子请求将继承这个新头域。

这是一个设置 `Content-Type` 头域的例子：

```
ngx.req.set_header("Content-Type", "text/css")
```

`header_value` 可以是一个值列表，例如：

```
ngx.req.set_header("Foo", { "a", "abc" })
```

将产生两个头域：

```

Foo: a
Foo: abc

```

如果有老的 `Foo` 头域，将被覆盖。当 `header_value` 参数是 `nil`，头域将被移除：

```
ngx.req.set_header("X-Foo", nil)
```

等同于

```
ngx.req.clear_header("X-Foo")
```

## 27. ngx.req.clear\_header

语法:

```
ngx.req.clear_header(header_name)
```

上下文: set\_by\_lua\*、rewrite\_by\_lua\*、access\_by\_lua\*、content\_by\_lua\*、header\_filter\_by\_lua\*、body\_filter\_by\_lua\*。

说明: 清除当前请求的 header\_name 头域。所有的子请求也将受到影响, 对应的头域也被清除。

## 28. ngx.req.read\_body

语法:

```
ngx.req.read_body()
```

上下文: rewrite\_by\_lua\*、access\_by\_lua\*、content\_by\_lua\*。

说明: 同步地读取客户端请求包体, 但不阻塞 Nginx 事件循环。

```
ngx.req.read_body()
```

```
local args = ngx.req.get_post_args()
```

如果请求包体已经通过打开 lua\_need\_request\_body 或其他模块读取, 本函数会立即返回而并不执行。如果包体已经被 ngx.req.discard\_body 或其他模块丢弃, 本函数也是立即返回而不执行。当读取数据的时候发生任意错误, 本方法将抛出 Lua 异常或使用 500 状态码立即退出。如果请求包体通过本函数读取, 则后续还可以通过 ngx.req.get\_body\_data 获取。另外, 包体数据可以通过 ngx.req.get\_body\_file 缓存到临时文件中。这取决于:

1) 当前请求包体尺寸是否大于 client\_body\_buffer\_size。

2) client\_body\_in\_file\_only 是否被打开。

在当前请求有一个包体, 而我们正好不需要包体数据的时候, 使用 ngx.req.discard\_body 函数可以解决我们的需求, 同时避免中断连接, 同时可以保证 HTTP 1.1 的心跳正常进行或 HTTP 1.1 的正常流转。

## 29. ngx.req.discard\_body

语法:

```
ngx.req.discard_body()
```

上下文: rewrite\_by\_lua\*、access\_by\_lua\*、content\_by\_lua\*。

说明: 丢弃请求的包体, 在不需要使用包体的情况下, 立即将连接上的包体丢弃。这个函数是一个异步调用, 会立即返回。如果请求包体已经读取, 这个函数立即返回, 不做

任何处理。

### 30. ngx.req.get\_body\_data

语法:

```
data = ngx.req.get_body_data()
```

上下文: `rewrite_by_lua*`、`access_by_lua*`、`content_by_lua*`、`log_by_lua*`。

说明: 获取内存中的包体数据, 返回一个容纳解析好查询参数的 Lua 字符串。使用 `ngx.req.get_post_args` 会返回一个 Lua 表。

下面情况中, 函数将返回 `nil`:

- 请求的包体还未读取。
- 请求的包体被读取到磁盘的临时文件中。
- 请求包体是零长度。

如果请求包体还没有读取, 可以使用 `ngx.req.read_body` 进行读取 (或打开 `lua_need_request_body` 强制模块读取请求包体, 但不推荐这个方法)。如果包体已经被读取并存放到磁盘文件中, 需要尝试调用 `ngx.req.get_body_file` 函数替代。要强制使用内存请求包体, 需要尝试设置 `client_body_buffer_size` 与 `client_max_body_size` 值一样。

注意, 使用本函数代替 `ngx.var.request_body` 或 `ngx.var.echo_request_body` 更有效, 因为它可以保存一个动态内存申请, 使用一个数据副本。

### 31. ngx.req.get\_body\_file

语法:

```
file_name = ngx.req.get_body_file()
```

上下文: `rewrite_by_lua*`、`access_by_lua*`、`content_by_lua*`。

说明: 获取请求包体数据的临时缓存文件名。如果请求包体没有读取或没有被读到内存中, 会返回 `nil`。这个文件是只读的, 由 Nginx 内存池负责清除。不可以在 Lua 代码中手工编辑、改名或删除。

如果请求包体仍然没有被读取, 那么首先调用 `ngx.req.read_body` (或打开 `lua_need_request_body` 强制模块读取请求包体, 但这个方法并不推荐)。如果请求包体被读取到内存中了, 尝试使用 `ngx.req.get_body_data` 函数代替。要强制使用文件化的请求包体, 尝试打开 `client_body_in_file_only` 指令。

### 32. ngx.req.set\_body\_data

语法:

```
ngx.req.set_body_data(data)
```

上下文: `rewrite_by_lua*`、`access_by_lua*`、`content_by_lua*`。

说明: 设置当前请求使用内存中的 `data` 数据。如果当前请求包体还没有读取, 那么会

被丢弃。当前请求包体已经被读取进内存或放到一个磁盘文件中，旧请求包体内存将被释放，磁盘文件会被立即清除。

### 33. ngx.req.set\_body\_file

语法：

```
ngx.req.set_body_file(file_name, auto_clean?)
```

上下文：rewrite\_by\_lua\*、access\_by\_lua\*、content\_by\_lua\*。

说明：将当前请求包体设置为使用 file\_name 指定的文件内容。

auto\_clean 是可选参数，如果设置为 true，那么请求处理完毕或下一次本函数被调用或 ngx.req.set\_body\_data 在相同的请求上被调用，本文件将被删除。默认 auto\_clean 是 false。请确保 file\_name 指定的文件存在并且可以被 Nginx 工作进程读取。要设置正确的权限以避免 Lua 异常错误。如果当前请求包体还没有被读取，则包体将被丢弃。如果当前请求包体已经被读取进内存或一个磁盘文件，那么旧请求包体内存将立即被释放和清理。

### 34. ngx.req.init\_body

语法：

```
ngx.req.init_body(buffer_size?)
```

上下文：set\_by\_lua\*、rewrite\_by\_lua\*、access\_by\_lua\*、content\_by\_lua\*。

说明：为当前请求创建一个新的空请求包体，这个缓冲在后续请求包体数据到来时，可以使用 ngx.req.append\_body 和 ngx.req.finish\_body 写入。如果指定了 buffer\_size 参数，则包体尺寸也指定了。如果未指定本参数，则使用 client\_body\_buffer\_size 指令的数值代替。当请求包体不能在内存缓冲区里容纳时，像 Nginx 核心处理标准请求包体那样将数据刷写进一个临时文件。

所有的数据都写入后，调用 ngx.req.finish\_body 是非常重要的。当本函数和 ngx.req.socket 一起使用时，需要在本函数之前调用 ngx.req.socket，否则会得到一个“request body already exists”的错误消息。

经常这样使用本函数：

```
ngx.req.init_body(128*1024) -- buffer is 128KB
for chunk in next_data_chunk() do
    ngx.req.append_body(chunk) -- each chunk can be 4KB
end
ngx.req.finish_body()
```

本函数可以和 ngx.req.append\_body、ngx.req.finish\_body、ngx.req.socket 一起使用，用 Lua 脚本实现输入过滤器（在 rewrite\_by\_lua\* 和 access\_by\_lua\* 中使用），可以被其他上下文处理器或上游模块使用，如 ngx\_http\_proxy\_module 和 ngx\_http\_fastcgi\_module 上游模块。

### 35. ngx.req.append\_body

语法：

```
ngx.req.append_body(data_chunk)
```

上下文: `set_by_lua*`、`rewrite_by_lua*`、`access_by_lua*`、`content_by_lua*`。

说明: 把 `data_chunk` 指定的新数据追加到 `ngx.req.init_body` 初始化请求包体后。当数据可以不再缓冲在内存中时, 会与 Nginx 核心处理标准请求包体一样, 被刷写进一个临时文件。当所有的数据被追加到包体后, 需要调用 `ngx.req.finish_body`。

本函数可以和 `ngx.req.init_body`、`ngx.req.finish_body`、`ngx.req.socket` 一起使用, 实现输入过滤器 (在 `rewrite_by_lua*` 或 `access_by_lua*` 上下文中), 可以被其他 Nginx 上下文处理器或上游模块 (如 `ngx_http_proxy_module` 和 `ngx_http_fastcgi_module`) 使用。

### 36. ngx.req.finish\_body

语法:

```
ngx.req.finish_body()
```

上下文: `set_by_lua*`、`rewrite_by_lua*`、`access_by_lua*`、`content_by_lua*`。

说明: 结束 `ngx.req.init_body` 和 `ngx.req.append_body` 调用创建的请求包体构造过程。本函数可以和 `ngx.req.init_body`、`ngx.req.finish_body`、`ngx.req.socket` 一起使用, 实现输入过滤器 (在 `rewrite_by_lua*` 或 `access_by_lua*` 上下文中), 可以被其他 Nginx 上下文处理器或上游模块 (如 `ngx_http_proxy_module` 和 `ngx_http_fastcgi_module`) 使用。

### 37. ngx.req.socket

语法:

```
tcpsock, err = ngx.req.socket()
tcpsock, err = ngx.req.socket(raw)
```

上下文: `rewrite_by_lua*`、`access_by_lua*`、`content_by_lua*`。

说明: 返回下游连接只读的 `cosocket` 对象。任何错误下, 将返回 `nil`, 同时返回一个错误描述字符串。这个套接字对象经常用来以流式读取当前请求包体。不要打开 `lua_need_request_body` 指令, 不要把这个调用和 `ngx.req.read_body` 和 `ngx.req.discard_body` 混合使用。如果请求包体数据已经被 Nginx 核心预读进请求头缓冲区, `cosocket` 对象需要小心处理这种预读造成的数据丢失。暂时不支持分块读取。

从 v0.9.0 版本开始, 本函数接受一个可靠的布尔型 `raw` 参数。当这个参数为 `true` 时, 函数返回一个全双工的 `cosocket` 对象, 可以调用 `receive`、`receiveuntil` 和 `send` 方法。当 `raw` 参数为 `true` 时, 需要没有 `ngx.say`、`ngx.print` 或 `ngx.send_headers` 这种未定的调用存在。所以, 如果要输出信息, 需要在前面先行调用, 并且通过 `ngx.flush(true)` 结束数据待定状态, 确保没有待定状态的数据再调用 `ngx.req.socket(true)`。如果请求包体还没有准备好, 那么可以使用这个原始套接字读取请求包体。

可以使用 `ngx.req.socket(true)` 返回的原始套接字实现如 WebSocket 等协议, 或者提交自己的原始 HTTP 协议头和体。

参见第 23 章第 3 节 lua-resty-websocket 库，该节有一个真实的例子。

### 38. ngx.exec

语法：

```
ngx.exec(uri, args?)
```

上下文：rewrite\_by\_lua\*、access\_by\_lua\*、content\_by\_lua\*。

说明：使用 args 为参数重定向到 URI 上，与 echo-nginx-module 模块中的 echo\_exe 指令一样。ngx.exec 是内部重定向机制，与 ngx.location.capture 的子请求不同，子请求在请求中并行运行，而 exec 的重定向则中断当前请求处理，跳转到新的 location 上，等价于 rewrite：

```
rewrite regex replacement last;

ngx.exec('/some-location');
ngx.exec('/some-location', 'a=3&b=5&c=6');
ngx.exec('/some-location?a=3&b=5', 'c=6');
```

可选的第二个参数 args 可以用来指定扩展的 URI 查询参数，例如：

```
ngx.exec("/foo", "a=3&b=hello%20world")
```

也可以使用一个 Lua 表传递参数：

```
ngx.exec("/foo", { a = 3, b = "hello world" })
```

结果和上面的例子是一样的。

Lua 表的格式和 ngx.encode\_args 方法的格式相同，支持命名 location (Named locations)，但是 args 参数会被忽略。查询字符串会从上层 location 继承（如果有）。

GET /foo/file.php?a=hello，则下面例子将返回“hello”而不是“goodbye”：

```
location/foo {
    content_by_lua_block {
        ngx.exec("@bar", "a=goodbye");
    }
}

location@bar {
    content_by_lua_block {
        local args = ngx.req.get_uri_args()
        for key, val in pairs(args) do
            if key == "a" then
                ngx.say(val)
            end
        end
    }
}
```

注意，ngx.exec 和 ngx.redirect 不同，ngx.exec 是一个纯粹的内部重定向，不涉及额外的流量。也要注意，必须在 ngx.send\_headers 之前或确保应答包体已经通过 ngx.print or



ngx.say 发送出去才能中断当前处理过程。当本方法用在除了 header\_filter\_by\_lua\* 以外的上下文时，推荐使用本方法返回，以处理请求中断的处理过程。

### 39. ngx.redirect

语法：

```
ngx.redirect(uri, status?)
```

上下文：rewrite\_by\_lua\*、access\_by\_lua\*、content\_by\_lua\*。

说明：向 URI 发出一个 HTTP 301 或 302 重定向。可选参数 status 指定 HTTP 状态码。

支持下面的状态码：

- 301。
- 302 (默认)。
- 307。

302 状态码是默认值 (ngx.HTTP\_MOVED\_TEMPORARILY)。下面的例子假设当前服务器名字为 localhost，并且监听在 1984 端口上：

```
return ngx.redirect("/foo")
```

等价于：

```
return ngx.redirect("/foo", ngx.HTTP_MOVED_TEMPORARILY)
```

也支持重定向任意的外部 URL，例如：

```
return ngx.redirect("http://www.google.com")
```

status 参数也可以直接使用数字代码：

```
return ngx.redirect("/foo", 301)
```

本方法和 ngx\_http\_rewrite\_module 模块中 rewrite 指令的 redirect 关键字一样。例如 nginx.conf 片断：

```
rewrite^ /foo? redirect; # nginx config
```

等价于下面的 Lua 代码：

```
return ngx.redirect('/foo'); -- Lua code
```

下面的：

```
rewrite^ /foo? permanent; # nginx config
```

等价于：

```
return ngx.redirect('/foo', ngx.HTTP_MOVED_PERMANENTLY) -- Lua code
```

可以指定 URI 参数，例如：

```
return ngx.redirect('/foo?a=3&b=4')
```

注意，必须在 `ngx.send_headers` 之前或确保应答包体已经通过 `ngx.print` or `ngx.say` 发送出去才能中断当前处理过程。当本方法用在除了 `header_filter_by_lua*` 外的上下文时，推荐使用本方法返回，以处理请求中断的处理过程。

#### 40. ngx.send\_headers

语法：

```
ok, err = ngx.send_headers()
```

上下文：`rewrite_by_lua*`、`access_by_lua*`、`content_by_lua*`。

说明：发送一个应答头。

从 v0.8.3 版本开始，本函数成功时返回 1，否则返回 nil 和一个错误描述的字符串。

注意，一般情况下不需要手工发送应答头，`ngx_lua` 会在 `ngx.say` 或 `ngx.print` 或当 `content_by_lua*` 存在时自动发送应答头。

#### 41. ngx.headers\_sent

语法：

```
value = ngx.headers_sent
```

上下文：`set_by_lua*`、`rewrite_by_lua*`、`access_by_lua*`、`content_by_lua*`。

说明：如果应答包头被 `ngx_lua` 发送则返回 true，否则返回 false。

#### 42. ngx.print

语法：

```
ok, err = ngx.print(...)
```

上下文：`rewrite_by_lua*`、`access_by_lua*`、`content_by_lua*`。

说明：把参数级联后作为应答包体输出至 HTTP 客户端。如果还没有发送应答头，这个函数会先发送一个应答头。从 v0.8.3 版本开始，本函数返回 1 表示成功，返回 nil 表示失败，同时返回一个错误描述字符串。nil 值会输出 “nil” 字符串，布尔值会输出 “true” 或 “false” 字符串。允许使用嵌套的字符串数组，这些元素会被一个接一个地发送。

```
local table = {
    "hello, ",
    { "world: ", true, " or ", false,
      { ": ", nil } }
}
ngx.print(table)
```

将得到输出：

```
hello, world: true or false: nil
```

非数组表参数会引发一个 Lua 异常。`ngx.null` 常量会得到一个 “null” 字符串输出。这是一个异步调用，并且会立即返回，不等待数据写入系统发送缓冲区。如果运行在同步模

式，需要在调用后调用 `ngx.flush(true)`。可以显式引发流的输出。

注意，`ngx.print` 和 `ngx.say` 总是会调用 Nginx 输出包体过滤器链，这是一个昂贵的操作。所以，不要在一个高效循环中调用这两个函数，可在 Lua 中缓冲数据，节省调用消耗。

### 43. ngx.say

语法：

```
ok, err = ngx.say(...)
```

上下文：rewrite\_by\_lua\*、access\_by\_lua\*、content\_by\_lua\*。

说明：跟 `ngx.print` 一样，只是每次提交一行数据（数据结尾有换行符）。

### 44. ngx.log

语法：

```
ngx.log(log_level, ...)
```

上下文：init\_by\_lua\*、init\_worker\_by\_lua\*、set\_by\_lua\*、rewrite\_by\_lua\*、access\_by\_lua\*、content\_by\_lua\*、header\_filter\_by\_lua\*、body\_filter\_by\_lua\*、log\_by\_lua\*、ngx.timer.\*、balancer\_by\_lua\*、ssl\_certificate\_by\_lua\*、ssl\_session\_fetch\_by\_lua\*、ssl\_session\_store\_by\_lua\*。

说明：使用给定的日志级别，将参数追加到 `error.log` 文件中。接收 `nil` 参数，输出的时候显示 “nil”，布尔值显示 “true” 和 “false”，`ngx.null` 常量显示 “null”。`log_level` 参数的取值是 `ngx.ERR` 和 `ngx.WARN` 等常量，参见本章第 7 节 log 级别常量。

错误消息被 Nginx 核心限制为 2048B。限制部分包含了时间、换行符等。如果消息超过了这个限制，超过的部分会被截断。这个限制可以手工修改，在 `src/core/ngx_log.h` 中修改 `NGX_MAX_ERROR_STR` 宏。

### 45. ngx.flush

语法：

```
ok, err = ngx.flush(wait?)
```

上下文：rewrite\_by\_lua\*、access\_by\_lua\*、content\_by\_lua\*。

说明：将输出强制输出到客户端。从 v0.3.1rc32 版本开始，接受一个可选的布尔型参数 `wait`（默认是 `false`）。使用默认值调用时，`flush` 发出一个异步的调用。`wait` 为 `true` 时，`flush` 切换为同步模式。同步模式下，函数直到输出数据都写到系统缓冲区或 `send_timeout` 设置的时间到期才会返回。注意，使用 Lua 例程机制意味着这个函数即使在同步模式下，也不能阻塞住 Nginx 事件循环。

当 `ngx.flush(true)` 在 `ngx.print` 或 `ngx.say` 之后立即调用，将使后面的函数运行在同步模式下，这对流式输出特别有用。

注意，当工作在 HTTP 1.0 输出缓冲模式下，`ngx.flush` 将变得不实用。从 v0.8.3 版本开

始, 这个函数返回 1 表示成功, 返回 nil 和 err 错误描述信息表示失败。

#### 46. ngx.exit

语法:

```
ngx.exit(status)
```

上下文: `rewrite_by_lua*`、`access_by_lua*`、`content_by_lua*`、`header_filter_by_lua*`、`ngx.timer.*`、`balancer_by_lua*`、`ssl_certificate_by_lua*`、`ssl_session_fetch_by_lua*`、`ssl_session_store_by_lua*`。

说明: 当 `status >= 200` (`ngx.HTTP_OK`) 时, 将中断当前请求运行, 并将状态码返回给 Nginx。当 `status == 0` (`ngx.OK`) 时, 只会退出当前的上下文处理器 (或内容处理器, 如果使用了 `content_by_lua*` 命令), 继续运行当前请求下面的上下文过程 (如果有)。`status` 参数的取值可以是 `ngx.ERR`、`ngx.HTTP_NOT_FOUND`、`ngx.HTTP_MOVED_TEMPORARILY`、`ngx.OK` 或其他 HTTP status 常量。

如果需要返回一个自定义内容的错误页, 可以使用下面的代码片段:

```
ngx.status= ngx.HTTP_GONE
ngx.say("This is our own content")
-- to cause quit the whole request rather than the current phase handler
ngx.exit(ngx.HTTP_OK)
```

代码效果如下:

```
$ curl -i http://localhost/test
HTTP/1.1 410 Gone
Server: nginx/1.0.6
Date: Thu, 15 Sep 2011 00:51:48 GMT
Content-Type: text/plain
Transfer-Encoding: chunked
Connection: keep-alive
```

```
This is our own content
```

可以直接使用数值型错误码, 例如:

```
ngx.exit(501)
```

注意, 当本方法接受所有 HTTP 状态常量的时候, 仅接受核心常量中的 `NGX_OK` 和 `NGX_ERROR`。使用本方法中断当前请求的处理过程时, 推荐和 `return` 组合使用, 如 `return ngx.exit(...)`, 强化请求进程被退出的事实。当用于 `header_filter_by_lua` 和 `ssl_session_store_by_lua*` 上下文中时, `ngx.exit()` 是一个异步操作, 立即返回。使用和 `return` 组合的方法可以改变为同步, 使退出行为更稳定。

#### 47. ngx.eof

语法:

```
ok, err = ngx.eof()
```

上下文: `rewrite_by_lua*`、`access_by_lua*`、`content_by_lua*`。

说明: 在输出流里明确指出应答的结尾。在 HTTP 1.1 输出编码块中, 这将触发 Nginx 核心发送 “last chunk”。当禁用了 HTTP1.1 的 `keepalive` 特性后可以通过调用 `ngx.eof()` 使客户端主动断开连接, 这个技巧可以用来做一些后台工作而不需要 HTTP 客户端等待连接。

参见下面的例子:

```
location= /async {
    keepalive_timeout 0;
    content_by_lua_block {
        ngx.say("got the task!")
        ngx.eof() -- 下游 HTTP 客户端将在这个点关闭连接。
        -- access MySQL、PostgreSQL、Redis、Memcached 等
    }
}
```

当创建子请求来请求在其他 location 配置的上游模块时, 应该配置这些上游模块来忽略客户端连接的中断, 如果默认不是忽略的话。例如, 默认的标准 `ngx_http_proxy_module` 模块会在客户端断开连接后立即同时终止子请求和主请求, 所以在模块 `ngx_http_proxy_module` 将 `proxy_ignore_client_abort` 设置为开启就十分重要:

```
proxy_ignore_client_abort on;
```

一个更好的方法是使用 `ngx.timer.at` 做后台工作。从 v0.8.3 版本开始, 这个函数返回 1 表示成功, 返回 nil 和包含错误提示信息的 `err` 表示失败。

`ngx.exit` 和 `ngx.eof` 的本质区别在于 `ngx.exit()` 用于中断当前操作, 不管是 `ngx_lua` 模块请求处理的当前阶段还是整个请求, 而 `ngx.eof` 只是用于结束响应流的输出, 中断 HTTP 连接, 后面的代码逻辑还会继续在服务端执行, 而且 `ngx.eof` 支持运行的上下文比 `ngx.exit` 少很多, `ngx.eof` 有返回值, `ngx.exit` 没有, 因为请求已经结束。

## 48. ngx.sleep

语法:

```
ngx.sleep(seconds)
```

上下文: `rewrite_by_lua*`、`access_by_lua*`、`content_by_lua*`、`ngx.timer.*`、`ssl_certificate_by_lua*`、`ssl_session_fetch_by_lua*`。

说明: 休眠时间 (秒), 释放 CPU, 不阻塞进程。可以指定的时间精度是 0.001 秒 (1 毫秒)。在这个精度和机制下, 这个方法对 Nginx 时钟非常有用。从 0.7.20 版本, 0 可以作为参数指定。

## 49. ngx.escape\_uri

语法:

```
newstr = ngx.escape_uri(str)
```

上下文: `init_by_lua*`、`init_worker_by_lua*`、`set_by_lua*`、`rewrite_by_lua*`、`access_by_lua*`、

content\_by\_lua\*、header\_filter\_by\_lua\*、body\_filter\_by\_lua\*、log\_by\_lua\*、ngx.timer.\*、balancer\_by\_lua\*、ssl\_certificate\_by\_lua\*、ssl\_session\_fetch\_by\_lua\*、ssl\_session\_store\_by\_lua\*。

说明：把 str 进行 URI 格式编码。

## 50. ngx.unescape\_uri

语法：

```
newstr = ngx.unescape_uri(str)
```

上下文：init\_by\_lua\*、init\_worker\_by\_lua\*、set\_by\_lua\*、rewrite\_by\_lua\*、access\_by\_lua\*、content\_by\_lua\*、header\_filter\_by\_lua\*、body\_filter\_by\_lua\*、log\_by\_lua\*、ngx.timer.\*、balancer\_by\_lua\*、ssl\_certificate\_by\_lua\*。

说明：对 str 进行 URI 解码。例如：

```
ngx.say(ngx.unescape_uri("b%20r56+7"))
```

输出：

```
b r56 7
```

## 51. ngx.encode\_args

语法：

```
str = ngx.encode_args(table)
```

上下文：set\_by\_lua\*、rewrite\_by\_lua\*、access\_by\_lua\*、content\_by\_lua\*、header\_filter\_by\_lua\*、body\_filter\_by\_lua\*、log\_by\_lua\*、ngx.timer.\*、balancer\_by\_lua\*、ssl\_certificate\_by\_lua\*。

说明：把 Lua 表编码为查询参数字符串。例如：

```
ngx.encode_args({foo =3, ["b r"] ="hello world"})
```

输出：

```
foo=3&b%20r=hello%20world
```

表索引必须是 Lua 字符串。支持多值查询参数。只需要对多值的变量使用一个 Lua 表，例如：

```
ngx.encode_args({baz = {32, "hello"}})
```

获得输出：

```
baz=32&baz=hello
```

如果返回的值表是空的，表示值是 nil。同样支持布尔值，例如：

```
ngx.encode_args({a =true, b =1})
```

输出：

```
a&b=1
```

如果参数值是 false，等同于 nil 值。

## 52. ngx.decode\_args

语法：

```
table = ngx.decode_args(str, max_args?)
```

上下文: set\_by\_lua\*、rewrite\_by\_lua\*、access\_by\_lua\*、content\_by\_lua\*、header\_filter\_by\_lua\*、body\_filter\_by\_lua\*、log\_by\_lua\*、ngx.timer.\*、balancer\_by\_lua\*、ssl\_certificate\_by\_lua\*、ssl\_session\_fetch\_by\_lua\*、ssl\_session\_store\_by\_lua\*。

说明：将查询字符串 str 解码为 Lua 表，这是 ngx.encode\_args 的反向操作。可靠参数 max\_args 用于指定从 str 中解析的最大参数数量。默认的值是 100（包括多值的相同名字），额外的 URI 参数被静默地丢弃，以防止 Dos 攻击。

max\_args 可以设置为 0，表示取消限制，以处理所有接收到的参数：

```
local args = ngx.decode_args(str, 0)
```

不建议移除最大值限制。

## 53. ngx.encode\_base64

语法：

```
newstr = ngx.encode_base64(str, no_padding?)
```

上下文: set\_by\_lua\*、rewrite\_by\_lua\*、access\_by\_lua\*、content\_by\_lua\*、header\_filter\_by\_lua\*、body\_filter\_by\_lua\*、log\_by\_lua\*、ngx.timer.\*、balancer\_by\_lua\*、ssl\_certificate\_by\_lua\*、ssl\_session\_fetch\_by\_lua\*、ssl\_session\_store\_by\_lua\*。

说明：将 str 进行 Base64 编码。从 0.9.16 版本开始，布尔型 no\_padding 参数可以指定控制是否在结果尾增加一个 Base64 填充（默认是 false，表示要增加填充）。

## 54. ngx.decode\_base64

语法：

```
newstr = ngx.decode_base64(str)
```

上下文: set\_by\_lua\*、rewrite\_by\_lua\*、access\_by\_lua\*、content\_by\_lua\*、header\_filter\_by\_lua\*、body\_filter\_by\_lua\*、log\_by\_lua\*、ngx.timer.\*、balancer\_by\_lua\*、ssl\_certificate\_by\_lua\*、ssl\_session\_fetch\_by\_lua\*、ssl\_session\_store\_by\_lua\*。

说明：将 str 进行 Base64 解码，还原为原始格式。如果返回 nil，则表示格式不对。

## 55. ngx.crc32\_short

语法：

```
intval = ngx.crc32_short(str)
```

上下文: `set_by_lua*`、`rewrite_by_lua*`、`access_by_lua*`、`content_by_lua*`、`header_filter_by_lua*`、`body_filter_by_lua*`、`log_by_lua*`、`ngx.timer.*`、`balancer_by_lua*`、`ssl_certificate_by_lua*`、`ssl_session_fetch_by_lua*`、`ssl_session_store_by_lua*`。

说明: 该方法主要用于计算给定字符串 `str` 的循环校验码 (Cyclic Redundancy Code, CRC) 的摘要, 计算出来的结果是一个很大的整数。本方法在短的 `str` 输入 (小于 30-60 字节) 情况下性能良好。本方法只是 `ngx_crc32_short` 的一个 `ngx_lua` 封装。本方法与 `ngx_crc32_long` 基本一样, 只是输入数据长度不同。

## 56. ngx\_crc32\_long

语法:

```
intval = ngx_crc32_long(str)
```

上下文: `set_by_lua*`、`rewrite_by_lua*`、`access_by_lua*`、`content_by_lua*`、`header_filter_by_lua*`、`body_filter_by_lua*`、`log_by_lua*`、`ngx.timer.*`、`balancer_by_lua*`、`ssl_certificate_by_lua*`、`ssl_session_fetch_by_lua*`、`ssl_session_store_by_lua*`。

说明: 该方法主要用于计算给定字符串 `str` 的 CRC 的摘要, 计算出来的结果是一个很大的整数。本方法在一个长 `str` 输入 (大于 30-60 个字节) 的情况下性能良好。这个方法是 `ngx_crc32_long` 函数的一个封装。

## 57. ngx\_hmac\_sha1

语法:

```
digest = ngx_hmac_shal(secret_key, str)
```

上下文: `set_by_lua*`、`rewrite_by_lua*`、`access_by_lua*`、`content_by_lua*`、`header_filter_by_lua*`、`body_filter_by_lua*`、`log_by_lua*`、`ngx.timer.*`、`balancer_by_lua*`、`ssl_certificate_by_lua*`、`ssl_session_fetch_by_lua*`、`ssl_session_store_by_lua*`。

说明: 该方法主要用于计算输入字符串 `str` 的 HMAC-SHA1 的摘要, 并根据 `secret_key` 对结果进行转换, 计算后得到的结果是二进制格式的, 可以通过 `ngx.encode_base64` 转换成非二进制格式的字符串。例如:

```
local key = "thisisverysecretstuff"
local src = "some string we want to sign"
local digest = ngx_hmac_shal(key, src)
ngx.say(ngx.encode_base64(digest))
```

输出:

```
R/pvxzHC4NLtj7S+kxFg/NePTmk=
```

这个 API 需要在 Nginx 里使能 OpenSSL (`./configure --with-http_ssl_moudule`)。

## 58. ngx\_md5

语法:



```
digest = ngx.md5(str)
```

上下文: set\_by\_lua\*、rewrite\_by\_lua\*、access\_by\_lua\*、content\_by\_lua\*、header\_filter\_by\_lua\*、body\_filter\_by\_lua\*、log\_by\_lua\*、ngx.timer.\*、balancer\_by\_lua\*、ssl\_certificate\_by\_lua\*、ssl\_session\_fetch\_by\_lua\*、ssl\_session\_store\_by\_lua\*。

说明: 返回字符串 str 的 MD5 摘要的十六进制表示。例如:

```
location= /md5 {
    content_by_lua_block { ngx.say(ngx.md5("hello")) }
}
```

输出:

```
5d41402abc4b2a76b9719d911017c592
```

## 59. ngx.md5\_bin

语法:

```
digest = ngx.md5_bin(str)
```

上下文: set\_by\_lua\*、rewrite\_by\_lua\*、access\_by\_lua\*、content\_by\_lua\*、header\_filter\_by\_lua\*、body\_filter\_by\_lua\*、log\_by\_lua\*、ngx.timer.\*、balancer\_by\_lua\*、ssl\_certificate\_by\_lua\*、ssl\_session\_fetch\_by\_lua\*、ssl\_session\_store\_by\_lua\*。

说明: 该方法将返回字符串 str 的 MD5 摘要的二进制格式, 可以通过 ngx.encode\_base64 方法转换成非二进制格式的字符串, 或者直接使用 ngx.md5 方法。

## 60. ngx.sha1\_bin

语法:

```
digest = ngx.shal_bin(str)
```

上下文: set\_by\_lua\*、rewrite\_by\_lua\*、access\_by\_lua\*、content\_by\_lua\*、header\_filter\_by\_lua\*、body\_filter\_by\_lua\*、log\_by\_lua\*、ngx.timer.\*、balancer\_by\_lua\*、ssl\_certificate\_by\_lua\*、ssl\_session\_fetch\_by\_lua\*、ssl\_session\_store\_by\_lua\*。

说明: 该方法返回字符串 str 二进制格式的 SHA-1 摘要。本函数需要 Nginx 支持 SHA-1 算法 (意味着需要在 Nginx 中安装或使能 OpenSSL)。

例如:

```
location /encryption {
    content_by_lua '
        local crc_32s, crc_32l
        local key = "it is my secret"
        local str = "encrypted hello yuefei"
        crc_32s = ngx.crc32_short(str)
        crc_32l = ngx.crc32_long(str)
```

```

local hmac = ngx.hmac_shal(key, str)
local md5 = ngx.md5(str)
local md5_bin = ngx.md5(str)
local sha1_bin = ngx.shal_bin(str)
ngx.say("crc_32_short:", crc_32s, ", crc_32_long: ", crc_32l)
ngx.say("hmac: ", ngx.encode_base64(hmac))
ngx.say("md5: ", md5, ", md5_bin: ", ngx.encode_base64(md5_bin))
ngx.say("shal_bin: ", ngx.encode_base64(shal_bin))

';
}

```

输出:

```

crc_32_short:1560312840,crc_32_long: 1560312840
hmac: 1gpvtAliGFzfSqSD32Sz04/3PiM=
md5: b80a89b331b307dbef83e2eb90c43481, md5_bin: uAqJsZGzB9vvg+LrkMQ0gQ==
shal_bin: 2A/wkXlXjz1t3wmNxMUi3QuMP7c=

```

## 61. ngx.quote\_sql\_str

语法:

```
quoted_value = ngx.quote_sql_str(raw_value)
```

上下文: set\_by\_lua\*、rewrite\_by\_lua\*、access\_by\_lua\*、content\_by\_lua\*、header\_filter\_by\_lua\*、body\_filter\_by\_lua\*、log\_by\_lua\*、ngx.timer.\*、balancer\_by\_lua\*、ssl\_certificate\_by\_lua\*、ssl\_session\_fetch\_by\_lua\*、ssl\_session\_store\_by\_lua\*。

说明: 根据 MySQL 查询字符串标准返回一个 SQL 字符串。

## 62. ngx.today

语法:

```
str = ngx.today()
```

上下文: init\_worker\_by\_lua\*、set\_by\_lua\*、rewrite\_by\_lua\*、access\_by\_lua\*、content\_by\_lua\*、header\_filter\_by\_lua\*、body\_filter\_by\_lua\*、log\_by\_lua\*、ngx.timer.\*、balancer\_by\_lua\*、ssl\_certificate\_by\_lua\*、ssl\_session\_fetch\_by\_lua\*、ssl\_session\_store\_by\_lua\*。

说明: 从 Nginx 缓冲时间中返回当前日期 (格式 yyyy-mm-dd), 返回的时间是本地时间。

## 63. ngx.time

语法:

```
secs = ngx.time()
```

上下文: init\_worker\_by\_lua\*、set\_by\_lua\*、rewrite\_by\_lua\*、access\_by\_lua\*、content\_by\_lua\*、header\_filter\_by\_lua\*、body\_filter\_by\_lua\*、log\_by\_lua\*、ngx.timer.\*、balancer\_by\_lua\*、ssl\_certificate\_by\_lua\*、ssl\_session\_fetch\_by\_lua\*、ssl\_session\_store\_by\_lua\*。

说明：从 Nginx 缓冲的时间（与 Lua 的 `date` 库不同，无系统调用）返回从纪元开始到当前流逝的秒数。更新 Nginx 时间缓存，首先需要调用 `ngx.update_time`。

#### 64. ngx.now

语法：

```
secs = ngx.now()
```

上下文：init\_worker\_by\_lua\*、set\_by\_lua\*、rewrite\_by\_lua\*、access\_by\_lua\*、content\_by\_lua\*、header\_filter\_by\_lua\*、body\_filter\_by\_lua\*、log\_by\_lua\*、ngx.timer.\*、balancer\_by\_lua\*、ssl\_certificate\_by\_lua\*、ssl\_session\_fetch\_by\_lua\*、ssl\_session\_store\_by\_lua\*。

说明：返回一个浮点型的流逝秒数值（小时部分表示毫秒）。本调用无系统调用，调用的是缓冲的时间。可以先调用 `ngx.update_time` 强制同步时间缓存。

#### 65. ngx.update\_time

语法：

```
ngx.update_time()
```

上下文：init\_worker\_by\_lua\*、set\_by\_lua\*、rewrite\_by\_lua\*、access\_by\_lua\*、content\_by\_lua\*、header\_filter\_by\_lua\*、body\_filter\_by\_lua\*、log\_by\_lua\*、ngx.timer.\*、balancer\_by\_lua\*、ssl\_certificate\_by\_lua\*、ssl\_session\_fetch\_by\_lua\*、ssl\_session\_store\_by\_lua\*。

说明：强制更新 Nginx 当前时间缓存。这个调用要进行系统调用，所以有一些资源消耗，不要经常使用它。

#### 66. ngx.localtime

语法：

```
str = ngx.localtime()
```

上下文：init\_worker\_by\_lua\*、set\_by\_lua\*、rewrite\_by\_lua\*、access\_by\_lua\*、content\_by\_lua\*、header\_filter\_by\_lua\*、body\_filter\_by\_lua\*、log\_by\_lua\*、ngx.timer.\*、balancer\_by\_lua\*、ssl\_certificate\_by\_lua\*、ssl\_session\_fetch\_by\_lua\*、ssl\_session\_store\_by\_lua\*。

说明：从 Nginx 时间缓存返回当前的时间（本地时间，格式为 `yyyy-mm-dd hh:mm:ss`，与 Lua 的 `date` 函数不同，本方法无系统调用）。

#### 67. ngx.utctime

语法：

```
str = ngx.utctime()
```

上下文：init\_worker\_by\_lua\*、set\_by\_lua\*、rewrite\_by\_lua\*、access\_by\_lua\*、content\_by\_lua\*、header\_filter\_by\_lua\*、body\_filter\_by\_lua\*、log\_by\_lua\*、ngx.timer.\*、balancer\_by\_lua\*、ssl\_certificate\_by\_lua\*、ssl\_session\_fetch\_by\_lua\*、ssl\_session\_store\_by\_lua\*。

说明：从 Nginx 时间缓存返回当前时间（UTC 时间，格式为 yyyy-mm-dd hh:mm:ss，与 Lua 的 date 函数不同，本方法无系统调用）。

## 68. ngx.cookie\_time

语法：

```
str = ngx.cookie_time(sec)
```

上下文：init\_worker\_by\_lua\*、set\_by\_lua\*、rewrite\_by\_lua\*、access\_by\_lua\*、content\_by\_lua\*、header\_filter\_by\_lua\*、body\_filter\_by\_lua\*、log\_by\_lua\*、ngx.timer.\*、balancer\_by\_lua\*、ssl\_certificate\_by\_lua\*、ssl\_session\_fetch\_by\_lua\*、ssl\_session\_store\_by\_lua\*。

说明：返回一个可以用做 cookie 期限的格式化时间字符串。sec 是以秒为单位的时间（与 ngx.time 返回的时间一样）。

```
ngx.say(ngx.cookie_time(1290079655))
-- yields "Thu, 18-Nov-10 11:27:35 GMT"
```

## 69. ngx.http\_time

语法：

```
str = ngx.http_time(sec)
```

上下文：init\_worker\_by\_lua\*、set\_by\_lua\*、rewrite\_by\_lua\*、access\_by\_lua\*、content\_by\_lua\*、header\_filter\_by\_lua\*、body\_filter\_by\_lua\*、log\_by\_lua\*、ngx.timer.\*、balancer\_by\_lua\*、ssl\_certificate\_by\_lua\*、ssl\_session\_fetch\_by\_lua\*、ssl\_session\_store\_by\_lua\*。

说明：返回一个 HTTP 头域可以使用的时间字符串（如 Last-Modified 头域）。参数 sec 是以秒为单位的时间（与 ngx.time 返回的一致）。

```
ngx.say(ngx.http_time(1290079655))
-- yields "Thu, 18 Nov 2010 11:27:35 GMT"
```

## 70. ngx.parse\_http\_time

语法：

```
sec = ngx.parse_http_time(str)
```

上下文：init\_worker\_by\_lua\*、set\_by\_lua\*、rewrite\_by\_lua\*、access\_by\_lua\*、content\_by\_lua\*、header\_filter\_by\_lua\*、body\_filter\_by\_lua\*、log\_by\_lua\*、ngx.timer.\*、balancer\_by\_lua\*、ssl\_certificate\_by\_lua\*、ssl\_session\_fetch\_by\_lua\*、ssl\_session\_store\_by\_lua\*。

说明：把 HTTP 时间字符串（与 ngx.http\_time 返回的数值一样）解析成秒。如果输入的字符串是错误的格式，则返回 nil。

```
local time = ngx.parse_http_time("Thu, 18 Nov 2010 11:27:35 GMT")
if time == nil then
    ...
end
```

## 71. ngx.is\_subrequest

语法:

```
value = ngx.is_subrequest
```

上下文: set\_by\_lua\*、rewrite\_by\_lua\*、access\_by\_lua\*、content\_by\_lua\*、header\_filter\_by\_lua\*、body\_filter\_by\_lua\*、log\_by\_lua\*。

说明: 如果当前请求是子请求, 则返回 true, 否则返回 false。

## 72. ngx.re.match

语法:

```
captures, err = ngx.re.match(subject, regex, options?, ctx?, res_table?)
```

上下文: init\_worker\_by\_lua\*、set\_by\_lua\*、rewrite\_by\_lua\*、access\_by\_lua\*、content\_by\_lua\*、header\_filter\_by\_lua\*、body\_filter\_by\_lua\*、log\_by\_lua\*、ngx.timer.\*、balancer\_by\_lua\*、ssl\_certificate\_by\_lua\*、ssl\_session\_fetch\_by\_lua\*、ssl\_session\_store\_by\_lua\*。

说明: 使用 Perl 兼容的正则表达式匹配 subject, 可选参数 options 是 regex 的参数。只有第一个出现的匹配会被返回, 如果没有匹配则返回 nil。遇到错误 (如一个错误的表达式) 或遇到 PCRE (Perl 库, 包括兼容的正则表达式库) 堆栈限制, 将返回 nil 值和错误描述 (err)。当发现一个匹配, captures 会返回一个 Lua 表, captures[0] 容纳的是第一个匹配的完整子串, capture[1] 保存的是用括号括起来的第一个子模式的结果, capture[2] 存放的是第二个子模式的结果, 依此类推。

```
local m, err = ngx.re.match("hello, 1234", "[0-9]+")
if m then
    -- m[0] == "1234"

else
    if err then
        ngx.log(ngx.ERR, "error: ", err)
        return
    end

    ngx.say("match not found")
end
```

上面例子中, 匹配的字符串是 1234, 因此 m[0] == "1234", 是没有用括号括起来的子模式, 因此, m[1]、m[2] 等均为 nil。

```
local m, err = ngx.re.match("hello, 1234", "([0-9])[0-9]+")
-- m[0] == "1234"
-- m[1] == "1"
```

因为有了子模式, 所以返回的 m[1] 有了值。

从 v0.7.14 版本后开始支持命名方式的捕获, 返回值存放在相同的 Lua 表中, 以 key-value 对存放。

```

local m, err = ngx.re.match("hello, 1234", "([0-9])(?<remaining>[0-9]+)")
-- m[0] == "1234"
-- m[1] == "1"
-- m[2] == "234"
-- m["remaining"] == "234"

```

未匹配的子域，将在结果里存放 false 值，索引是模式名。

```

local m, err = ngx.re.match("hello, world", "(world)|(hello)|(?<named>howdy)")
-- m[0] == "hello"
-- m[1] == nil
-- m[2] == "hello"
-- m[3] == nil
-- m["named"] == nil

```

上面例子中，第一个匹配的是 hello，所以 m[0] 是 hello，然后 3 个模式依次是，m[1] 是 world 没匹配上，m[2] 是 hello 匹配上，m[3] 没匹配上。所以，m[0] 是第一个匹配的字串。

可以使用 option 控制如何执行匹配操作，支持下面的选项字符：

- a: 锚定模式，只从头开始匹配。
- d: DFA 模式，或者称最长字符串匹配语义，需要 PCRE 6.0+ 支持。
- D: 允许重复命名的子模式，该选项需要 PCRE 8.12+ 支持。例如：

```

local m = ngx.re.match("hello, world", "(?<named>\w+), (?<named>\w+)", "D")
-- m["named"] == {"hello", "world"}

```

- i: 大小写不敏感模式。
- j: 启用 PCRE JIT 编译，需要 PCRE 8.21+ 支持，并且必须在编译时加上选项 --enable-jit，为了达到最佳性能，该选项总是应该和 'o' 选项搭配使用。
- J: 启用 PCRE JavaScript 的兼容模式，需要 PCRE 8.12+ 支持。
- m: 多行模式。
- o: 一次编译模式，启用 worker-process 级别的编译正则表达式的缓存。
- s: 单行模式。
- u: UTF-8 模式。该选项需要在编译 PCRE 库时加上 --enable-utf8 选项。
- U: 与 "u" 选项类似，但是禁止 PCRE 对 subject 字符串 UTF-8 有效性检查。
- x: 扩展模式。

这些选项可以组合使用。

```

local m, err = ngx.re.match("hello, world", "HEL LO", "ix")
-- m[0] == "hello"
local m, err = ngx.re.match("hello, 美好生活", "HELLO, (.{2})", "iu")
-- m[0] == "hello, 美好"
-- m[1] == "美好"

```

o 选项用于性能调优，因为正则表达式只会被编译一次，在工作进程级别缓存，在当前工作进程的所有请求间共享。正则缓存限制可以通过设置 lua\_regex\_cache\_max\_entries 指令实现。

可选参数 `ctx` 可以传入一个 Lua 表，传入的 Lua 表可以是一个空表，也可以是包含 `pos` 字段的 Lua 表。如果传入的是一个空的 Lua 表，那么，`ngx.re.match` 将会从 `subject` 字符串的起始位置开始匹配查找，查找到匹配串后，修改 `pos` 的值为匹配字符串的下一个位置的值，并将 `pos` 的值保存到 `ctx` 中，如果匹配失败，那么 `pos` 的值保持不变。如果传入的是一个非空的 Lua 表，即指定了 `pos` 的初值，那么 `ngx.re.match` 将会从指定的 `pos` 位置开始进行匹配，如果匹配成功了，修改 `pos` 值为匹配字符串下一个位置的值，并将 `pos` 值保存到 `ctx` 中，如果匹配失败，那么 `pos` 值保持不变。

```
local ctx = {}
local m, err = ngx.re.match("1234, hello", "[0-9]+", "", ctx)
-- m[0] = "1234"
-- ctx.pos == 5
local ctx = { pos = 2 }
local m, err = ngx.re.match("1234, hello", "[0-9]+", "", ctx)
-- m[0] = "34"
-- ctx.pos == 5
```

`ctx` 表参数和一个正则编辑器组合，可以用于在 `ngx.re.match` 之上构造自己的方法（但这显然意义不大）。注意，当传入了 `ctx` 参数时，`options` 不能可选，如果没有一个有意义的 `options`，必须在 `options` 用一个 Lua 空串 “”。必须在 Nginx 里使能 PCRE 库才能使用正则。

通过在 Nginx 或 OpenResty 的 `./configure` 脚本添加 `--with-debug` 使能 Nginx 日志调试，以确认 PCRE JIT 是否使能。将 `error_log` 指令设置为 `debug` 级别，下面的消息表示 PCRE JIT 已经使能了：

```
pcre JIT compiling result: 1
```

从 0.9.4 版本开始，这个函数已经接受第 5 个参数：`res_table`。为调用者提供了 Lua 表容纳所有捕获结果的能力。从 0.9.6 版本开始，调用者要自己保证这个表为空，这对回收表及 GC 和表申请的负载很有帮助。

### 73. ngx.re.find

语法：

```
from, to, err = ngx.re.find(subject, regex, options?, ctx?, nth?)
```

上下文：`init_worker_by_lua*`、`set_by_lua*`、`rewrite_by_lua*`、`access_by_lua*`、`content_by_lua*`、`header_filter_by_lua*`、`body_filter_by_lua*`、`log_by_lua*`、`ngx.timer.*`、`balancer_by_lua*`、`ssl_certificate_by_lua*`、`ssl_session_fetch_by_lua*`、`ssl_session_store_by_lua*`。

说明：和 `ngx.re.match` 基本相同，但是只返回匹配子串的开始位置 `index(from)` 和结束位置 `index(to)`。返回的索引从 1 开始，可以直接填充到 `string.sub`，获得匹配的子串。遇到错误（如一个错误的表达式）或遇到了 PCRE（Perl 库，包括兼容的正则表达式库）堆栈限制，将返回 `nil` 值和错误描述 `err`。

例如：

```

local s = "hello, 1234"
local from, to, err = ngx.re.find(s, "([0-9]+)", "jo")
if from then
    ngx.say("from: ", from)
    ngx.say("to: ", to)
    ngx.say("matched: ", string.sub(s, from, to))
else
    if err then
        ngx.say("error: ", err)
    end
    return
end
    ngx.say("not matched!")
end

```

输出:

```

from: 8
to: 11
matched: 1234

```

因为这个函数不创建新 Lua 字符串和新 Lua 表, 所以比 `ngx.re.match` 快很多, 应该在任何情况下优先使用。从 0.9.3 版本开始, 支持第五个可选参数, 可以指定第几个匹配的索引可以返回。当 `nth` 是 0 的时候 (默认值), 返回匹配的整体字符串; 当 `nth` 为 1 的时候, 只返回第一个子模式始末位置索引值; 当 `nth` 为 2 的时候, 只返回第二个子模式匹配的始末值, 依此类推。如果都没有匹配上, 则返回 2 个 `nil` 值。例如:

```

local str = "hello, 1234"
local from, to = ngx.re.find(str, "([0-9])([0-9]+)", "jo", nil, 2)
if from then
    ngx.say("matched 2nd submatch: ", string.sub(str, from, to)) -- yields
"234"
end

```

## 74. ngx.re.gmatch

语法:

```
iterator, err = ngx.re.gmatch(subject, regex, options?)
```

上下文: `init_worker_by_lua*`、`set_by_lua*`、`rewrite_by_lua*`、`access_by_lua*`、`content_by_lua*`、`header_filter_by_lua*`、`body_filter_by_lua*`、`log_by_lua*`、`ngx.timer.*`、`balancer_by_lua*`、`ssl_certificate_by_lua*`、`ssl_session_fetch_by_lua*`、`ssl_session_store_by_lua*`。

说明: 和 `ngx.re.match` 基本一致, 只是返回一个 Lua 迭代器, 用户可以使用迭代器遍历所有匹配的结果。如果匹配失败, 将会返回 `nil`; 如果匹配出现错误, 那么还会返回错误信息到 `err` 中。下面是一个演示基本使用方法的列子:

```

local iterator, err = ngx.re.gmatch("hello, world!", "([a-z]+)", "i")
ifnot iterator then
    ngx.log(ngx.ERR, "error: ", err)
    return
end

```



```

local m
m, err = iterator() -- m[0] == m[1] == "hello"
if err then
    ngx.log(ngx.ERR, "error: ", err)
    return
end

m, err = iterator() -- m[0] == m[1] == "world"
if err then
    ngx.log(ngx.ERR, "error: ", err)
    return
end

m, err = iterator() -- m == nil
if err then
    ngx.log(ngx.ERR, "error: ", err)
    return
end

```

通常我们把处理放到一个循环中:

```

local it, err = ngx.re.gmatch("hello, world!", "([a-z]+)", "i")
ifnot it then
    ngx.log(ngx.ERR, "error: ", err)
    return
end

whiletruedo
    local m, err = it()
    if err then
        ngx.log(ngx.ERR, "error: ", err)
        return
    end

    ifnot m then
        -- no match found (any more)
        break
    end

    -- found a match
    ngx.say(m[0])
    ngx.say(m[1])
end

```

可选的 options 参数和 ngx.re.match 方法的 options 用法一样。当前只允许在单个请求中使用返回的迭代器，不能把返回的迭代器赋值到持久存在的命名空间（如 Lua 包）。

这个方法同样需要在 Nginx 中使能 PCRE 库。

## 75. ngx.re.sub

语法:

```
newstr, n, err = ngx.re.sub(subject, regex, replace, options?)
```

使用域: `init_worker_by_lua*`、`set_by_lua*`、`rewrite_by_lua*`、`access_by_lua*`、`content_by_lua*`、`header_filter_by_lua*`、`body_filter_by_lua*`、`log_by_lua*`、`ngx.timer.*`、`balancer_by_lua*`、`ssl_certificate_by_lua*`、`ssl_session_fetch_by_lua*`、`ssl_session_store_by_lua*`。

说明: 在 `subject` 字符串中使用 `replace` 替换第一个匹配上 `regex` 的字符串。`options` 参数与 `ngx.re.match` 中的定义相同。成功替换后, `newstr` 返回新的字符串, `n` 存储替换的数量。如果失败了, 类似于表达式里有符号错误或 `<replace>` 错误, 将返回 `nil` 和 `err` 里的错误描述。若 `replace` 是一个字符串, 则会作为一个替换模板。例如:

```
local newstr, n, err = ngx.re.sub("hello, 1234", "([0-9])[0-9]", "${0}${1}")
if newstr then
-- newstr == "hello, [12][1]34"
-- n == 1
else
    ngx.log(ngx.ERR, "error: ", err)
return
end
```

在上面的例子中, `$0` 表示整个匹配的子串, `$1` 表示第一个子模式匹配的字串, 以此类推。可以用大括号 `{}` 将相应的 `0`、`1`、`2`……括起来, 以区分一般的数字:

```
local newstr, n, err = ngx.re.sub("hello, 1234", "[0-9]", "${0}00")
-- newstr == "hello, 100234"
-- n == 1
```

如果想在 `replace` 字符串中显示 `$` 符号, 可以用 `$` 进行转义 (不要用反斜杠 `\$` 对美元符号进行转义, 这种方法不会得到期望的结果):

```
local newstr, n, err = ngx.re.sub("hello, 1234", "[0-9]", "$$")
-- newstr == "hello, $234"
-- n == 1
```

如果 `replace` 是一个函数, 那么函数的参数是一个 “match table”, 而这个 “match table” 与 `ngx.re.match` 中的返回值 `captures` 是一样的, `replace` 这个函数根据 “match table” 产生用于替换的字符串, 例如:

```
local func =function (m)
return[".. m[0] .."][".. m[1] .."]
end
local newstr, n, err = ngx.re.sub("hello, 1234", "( [0-9] ) [0-9]", func, "x")
-- newstr == "hello, [12][1]34"
-- n == 1
```

通过函数形式返回的替换字符串中的符号 `$` 不再是特殊字符, 而只是被看作一个普通字符。

本方法同样需要 Nginx 使能 PCRE 库。

## 76. ngx.re.gsub

语法:

```
newstr, n, err = ngx.re.gsub(subject, regex, replace, options?)
```

上下文: init\_worker\_by\_lua\*、set\_by\_lua\*、rewrite\_by\_lua\*、access\_by\_lua\*、content\_by\_lua\*、header\_filter\_by\_lua\*、body\_filter\_by\_lua\*、log\_by\_lua\*、ngx.timer.\*、balancer\_by\_lua\*、ssl\_certificate\_by\_lua\*、ssl\_session\_fetch\_by\_lua\*、ssl\_session\_store\_by\_lua\*。

说明: 和 ngx.re.sub 一样, 只是不进行全局替换。例如:

```
local newstr, n, err = ngx.re.gsub("hello, world", "([a-z])[a-z]+", "[${0},${1}]",
                                   "i")

if newstr then
    -- newstr == "[hello,h], [world,w]"
    -- n == 2
else
    ngx.log(ngx.ERR, "error: ", err)
    return
end

local func =function (m)
    return[".. m[0] ..",".. m[1] .."]
end

local newstr, n, err = ngx.re.gsub("hello, world", "([a-z])[a-z]+", func, "i")
-- newstr == "[hello,h], [world,w]"
-- n == 2
```

本方法需要 Nginx 使能 PCRE 库。

## 77. ngx.shared.DICT

语法:

```
dict = ngx.shared.DICT
dict = ngx.shared[name_var]
```

上下文: init\_by\_lua\*、init\_worker\_by\_lua\*、set\_by\_lua\*、rewrite\_by\_lua\*、access\_by\_lua\*、content\_by\_lua\*、header\_filter\_by\_lua\*、body\_filter\_by\_lua\*、log\_by\_lua\*、ngx.timer.\*、balancer\_by\_lua\*、ssl\_certificate\_by\_lua\*、ssl\_session\_fetch\_by\_lua\*、ssl\_session\_store\_by\_lua\*。

说明: 获取定义在 lua\_shared\_dict 指令中的共享内存字典。共享内存字典是工作进程级的, 所以早已经被系统共享出来了。

可以使用下面的方法使用共享内存: get、get\_stale、set、safe\_set、add、safe\_add、replace、delete、incr、lpush、rpush、lpop、rpop、llen、flush\_all、flush\_expired、get\_keys。

在 ngx\_lua 模块中使用共享内存字典项相关 API 的前提条件是已经使用 lua\_shared\_dict 命令定义了一个字典项对象, 该命令的具体用法如下:

语法:

```
lua_shared_dict <name><size>
```

该命令用于定义一块名为 name 的共享内存空间, 内存大小为 size。通过该命令定义的

共享内存对象对于 Nginx 中所有工作进程都是可见的，当 Nginx 通过 reload 命令重启时，共享内存字典项会重新获取它的内容，当 Nginx 退出时，字典项的值将会丢失。下面是一个具体的例子：

```
http {
    lua_shared_dict dogs 10m;
    server {
        location /set {
            content_by_lua_block {
                local dogs = ngx.shared.dogs
                dogs:set("Jim", 8)
                ngx.say("STORED")
            }
        }

        location /get {
            content_by_lua_block {
                local dogs = ngx.shared.dogs
                ngx.say(dogs:get("Jim"))
            }
        }
    }
}
```

测试下：

```
$ curl localhost/set
STORED
```

```
$ curl localhost/get
8
```

```
$ curl localhost/get
8
```

当访问 /get 时，不管有多少个工作进程，总是输出 8。因为 dogs 辞典存在于共享内存中，所有的工作进程都可以看见并使用它。

共享辞典会一直保存它的内容，直到收到配置文件重载（收到 HUP 信号或 Nginx 通过 -s reload 重载）。当 Nginx 服务退出时，辞典里的数据将丢失。

在上面的例中，下面两种方法都可以：

```
local dogs =ngx.shared.dogs
local dogs = ngx.shared["dogs"]
```

获取了字典对象后，可以使用下面的方法进行的操作。

## 78. ngx.shared.DICT.get

语法：

```
value, flags = ngx.shared.DICT:get(key)
```

上下文: set\_by\_lua\*、rewrite\_by\_lua\*、access\_by\_lua\*、content\_by\_lua\*、header\_filter\_

by\_lua\*、body\_filter\_by\_lua\*、log\_by\_lua\*、ngx.timer.\*、balancer\_by\_lua\*、ssl\_certificate\_by\_lua\*、ssl\_session\_fetch\_by\_lua\*、ssl\_session\_store\_by\_lua\*。

说明：读取 key 对应的值。如果 key 不存在或过期了，将返回 nil 值。任何错误发生时，value 将返回 nil，flag 返回描述错误的字符串。返回的值携带原始插入时的数据类型，如 Lua 布尔值、数值、字符串。第一个参数必须是共享内存字典对象自己，例如：

```
local cats = ngx.shared.cats
local value, flags = cats.get(cats, "Marry")
```

或者使用 Lua 方法调用方法：

```
local cats = ngx.shared.cats
local value, flags = cats:get("Marry")
```

这两种方法一样。

返回列表中的 flags，是在 ngx.shared.DICT.set 方法中设置的值，默认值为 0。如果设置的 flags 为 0，那么这里不会返回 flags 值。

## 79. ngx.shared.DICT.get\_stale

语法：

```
value, flags, stale = ngx.shared.DICT:get_stale(key)
```

上下文：set\_by\_lua\*、rewrite\_by\_lua\*、access\_by\_lua\*、content\_by\_lua\*、header\_filter\_by\_lua\*、body\_filter\_by\_lua\*、log\_by\_lua\*、ngx.timer.\*、balancer\_by\_lua\*、ssl\_certificate\_by\_lua\*、ssl\_session\_fetch\_by\_lua\*、ssl\_session\_store\_by\_lua\*。

说明：和 get 方法一样，同时返回过期值。返回 3 个值，stale 表明值是否过期。

注意，一个过期的值不保证是有效的，所以，永远不要信任过期的值。

## 80. ngx.shared.DICT.set

语法：

```
success, err, forcible = ngx.shared.DICT:set(key, value, exptime?, flags?)
```

上下文：init\_by\_lua\*、set\_by\_lua\*、rewrite\_by\_lua\*、access\_by\_lua\*、content\_by\_lua\*、header\_filter\_by\_lua\*、body\_filter\_by\_lua\*、log\_by\_lua\*、ngx.timer.\*、balancer\_by\_lua\*、ssl\_certificate\_by\_lua\*、ssl\_session\_fetch\_by\_lua\*、ssl\_session\_store\_by\_lua\*。

说明：无条件向以共享内存为机制的字典插入一个 key-value 对，返回 3 个值。

- success：布尔值表示数据对是否存储。
- err：文本的错误信息，可以是“no memory”。
- forcible：布尔值，描述当空间不足时其他有效的元素是否被强制移走。

value 参数可以是 Lua 的布尔型、数值型、字符串或 nil。这些值类型会被字典保存，类型可以被 get 方法获取。

可选项 exptime 参数指定插入的数据的过期时间（以秒为单位）。时间精度分辨率是

0.001 秒。如果 `exptime` 值是 0 (默认值), 则元素永远不会过期。

选项 `flags` 参数指定了一个用户自定义标志一同保存在字典中, 后续可以获取。`flags` 作为一个 32 位的整形保存, 默认是 0。用户 `flags` 参数第一次出现在 v0.5.0rc2 版本中。

当为当前 `key-value` 项申请内存失败时, `set` 操作会根据 LRU 算法删除存储中已有的元素。注意, LRU 以过期时间为优先考虑对象。如果删除了 10 个元素, 剩下的空间仍然不足, 则 `err` 中将返回 “no memory” 信息, `success` 将为 `false`。如果通过 LRU 算法强行删除没有过期的元素使当前保存操作成功, 则 `forcible` 返回值为 `true`, 否则为 `false`。

第一个参数必须是字典对象本身, 例如:

```
local cats = ngx.shared.cats
local succ, err, forcible = cats.set(cats, "Marry", "it is a nice cat!")
```

或通过 Lua 的方法调用规则:

```
local cats = ngx.shared.cats
local succ, err, forcible = cats:set("Marry", "it is a nice cat!")
```

以上两种形式相同。注意, `key-value` 的设置内部是原子级的, 原子操作只会在 `set` 方法内部, 不会超过边界。

### 81. ngx.shared.DICT.safe\_set

语法:

```
ok, err = ngx.shared.DICT:safe_set(key, value, exptime?, flags?)
```

上下文: `init_by_lua*`、`set_by_lua*`、`rewrite_by_lua*`、`access_by_lua*`、`content_by_lua*`、`header_filter_by_lua*`、`body_filter_by_lua*`、`log_by_lua*`、`ngx.timer.*`、`balancer_by_lua*`、`ssl_certificate_by_lua*`、`ssl_session_fetch_by_lua*`、`ssl_session_store_by_lua*`。

说明: 和 `set` 方法相似, 只是在没有存储空间的时候永远不会覆盖还没过期的元素。这种情况下, 会立即返回 `nil` 和字符串 “no memory”。

### 82. ngx.shared.DICT.add

语法:

```
success, err, forcible = ngx.shared.DICT:add(key, value, exptime?, flags?)
```

使用域: `init_by_lua*`、`set_by_lua*`、`rewrite_by_lua*`、`access_by_lua*`、`content_by_lua*`、`header_filter_by_lua*`、`body_filter_by_lua*`、`log_by_lua*`、`ngx.timer.*`、`balancer_by_lua*`、`ssl_certificate_by_lua*`、`ssl_session_fetch_by_lua*`、`ssl_session_store_by_lua*`。

说明: 和 `set` 方法很像, 但只有 `key` 不存在的情况下将 `key-value` 对保存到字典内。如果 `key` 已经在字典内 (或还没过期), `success` 会返回 `false`, `err` 返回 “exists”。

### 83. ngx.shared.DICT.safe\_add

语法:

```
ok, err = ngx.shared.DICT:safe_add(key, value, exptime?, flags?)
```

上下文: init\_by\_lua\*、set\_by\_lua\*、rewrite\_by\_lua\*、access\_by\_lua\*、content\_by\_lua\*、header\_filter\_by\_lua\*、body\_filter\_by\_lua\*、log\_by\_lua\*、ngx.timer.\*、balancer\_by\_lua\*、ssl\_certificate\_by\_lua\*、ssl\_session\_fetch\_by\_lua\*、ssl\_session\_store\_by\_lua\*。

说明: 和 add 方法相似, 只是内存不足时永远不会覆盖 LRU 中没有过期的元素。这种情况下, 会立即返回 nil 和 “no memory” 字符串。

#### 84. ngx.shared.DICT.replace

语法:

```
success, err, forcible = ngx.shared.DICT:replace(key, value, exptime?, flags?)
```

上下文: init\_by\_lua\*、set\_by\_lua\*、rewrite\_by\_lua\*、access\_by\_lua\*、content\_by\_lua\*、header\_filter\_by\_lua\*、body\_filter\_by\_lua\*、log\_by\_lua\*、ngx.timer.\*、balancer\_by\_lua\*、ssl\_certificate\_by\_lua\*、ssl\_session\_fetch\_by\_lua\*、ssl\_session\_store\_by\_lua\*。

说明: 和 set 方法相似, 但是只是在 key 不存在的情况下将 key-value 对存储进字典。如果 key 不存在 (或者已经过期了), success 将返回 false, err 将返回 “not found”。

#### 85. ngx.shared.DICT.delete

语法:

```
ngx.shared.DICT:delete(key)
```

上下文: init\_by\_lua\*、set\_by\_lua\*、rewrite\_by\_lua\*、access\_by\_lua\*、content\_by\_lua\*、header\_filter\_by\_lua\*、body\_filter\_by\_lua\*、log\_by\_lua\*、ngx.timer.\*、balancer\_by\_lua\*、ssl\_certificate\_by\_lua\*、ssl\_session\_fetch\_by\_lua\*、ssl\_session\_store\_by\_lua\*。

说明: 无条件从共享内存字典中删除 key-value 对, 等于 ngx.shared.DICT:set(key, nil)。

#### 86. ngx.shared.DICT.incr

语法:

```
newval, err, forcible? = ngx.shared.DICT:incr(key, value, init?)
```

上下文: init\_by\_lua\*、set\_by\_lua\*、rewrite\_by\_lua\*、access\_by\_lua\*、content\_by\_lua\*、header\_filter\_by\_lua\*、body\_filter\_by\_lua\*、log\_by\_lua\*、ngx.timer.\*、balancer\_by\_lua\*、ssl\_certificate\_by\_lua\*、ssl\_session\_fetch\_by\_lua\*、ssl\_session\_store\_by\_lua\*。

说明: 使字典中 key 的值 (数值型) 按 value 增长。如果操作成功了, 则返回新的值, 否则返回 nil。当 key 不存在或已经过期的时候, 如果 init 没有指定或者数值为 nil, 则方法将返回 nil 和错误信息 “not found”。如果 init 参数是一个数值型值, 那么创建一个新的 key-value 对, 值是 init+value。和 add 方法一样, 当内存不足时会覆盖未过期的元素。

当 init 参数没有指定时, forcible 总是返回 nil。如果在内存不足时通过 LRU 删除未过期元素而存储成功, 则 forcible 值是 true。如果没有强制删除其他有效元素, 则 forcible 将

返回 false。如果原始的 value 是一个无效的 Lua 数值，则返回 nil 和 “not a number”。

value 参数和 inot 参数可以是任意有效的 Lua 数值，如负数和浮点数。本方法从 v0.3.1rc22 版本开始出现。init 参数在 v0.10.6 版本加入。

### 87. ngx.shared.DICT.lpush

语法：

```
length, err = ngx.shared.DICT:lpush(key, value)
```

上下文: init\_by\_lua\*、set\_by\_lua\*、rewrite\_by\_lua\*、access\_by\_lua\*、content\_by\_lua\*、header\_filter\_by\_lua\*、body\_filter\_by\_lua\*、log\_by\_lua\*、ngx.timer.\*、balancer\_by\_lua\*、ssl\_certificate\_by\_lua\*、ssl\_session\_fetch\_by\_lua\*、ssl\_session\_store\_by\_lua\*。

说明：在名为 key 的列表头部插入 value（数值型或字符串型），返回插入后的列表元素数目。如果 key 不存在，则首先创建一个空的列表。当 key 已经存在但不是一个列表时，则返回 nil 和 “value not a list” 错误信息。

### 88. ngx.shared.DICT.rpush

语法：

```
length, err = ngx.shared.DICT:rpush(key, value)
```

上下文: init\_by\_lua\*、set\_by\_lua\*、rewrite\_by\_lua\*、access\_by\_lua\*、content\_by\_lua\*、header\_filter\_by\_lua\*、body\_filter\_by\_lua\*、log\_by\_lua\*、ngx.timer.\*、balancer\_by\_lua\*、ssl\_certificate\_by\_lua\*、ssl\_session\_fetch\_by\_lua\*、ssl\_session\_store\_by\_lua\*。

说明：和 lpush 类似，只是在列表的尾部插入元素。

### 89. ngx.shared.DICT.lpop

语法：

```
val, err = ngx.shared.DICT:lpop(key)
```

上下文: init\_by\_lua\*、set\_by\_lua\*、rewrite\_by\_lua\*、access\_by\_lua\*、content\_by\_lua\*、header\_filter\_by\_lua\*、body\_filter\_by\_lua\*、log\_by\_lua\*、ngx.timer.\*、balancer\_by\_lua\*、ssl\_certificate\_by\_lua\*、ssl\_session\_fetch\_by\_lua\*、ssl\_session\_store\_by\_lua\*。

说明：删除并返回列表中第一个元素。如果 key 不存在，则返回 nil。当 key 已经存在但不是一个列表时，返回 nil 和 “value not a list” 错误信息。

### 90. ngx.shared.DICT.rpop

语法：

```
val, err = ngx.shared.DICT:rpoc(key)
```

上下文: init\_by\_lua\*、set\_by\_lua\*、rewrite\_by\_lua\*、access\_by\_lua\*、content\_by\_lua\*、header\_filter\_by\_lua\*、body\_filter\_by\_lua\*、log\_by\_lua\*、ngx.timer.\*、balancer\_by\_lua\*、



ssl\_certificate\_by\_lua\*、ssl\_session\_fetch\_by\_lua\*、ssl\_session\_store\_by\_lua\*。

说明：删除并返回 key 列表中的最后一个元素。如果 key 不存在，则返回 nil。当 key 已经存在但不是一个列表时，则返回 nil 和 “value not a list” 错误信息。

### 91. ngx.shared.DICT.llen

语法：

```
len, err = ngx.shared.DICT:llen(key)
```

上下文：init\_by\_lua\*、set\_by\_lua\*、rewrite\_by\_lua\*、access\_by\_lua\*、content\_by\_lua\*、header\_filter\_by\_lua\*、body\_filter\_by\_lua\*、log\_by\_lua\*、ngx.timer.\*、balancer\_by\_lua\*、ssl\_certificate\_by\_lua\*、ssl\_session\_fetch\_by\_lua\*、ssl\_session\_store\_by\_lua\*。

说明：返回 key 列表中元素的数量。如果 key 不存在，则中断并返回 0。当 key 已经存在但不是一个列表时，返回 nil 和 “value not a list” 错误信息。

### 92. ngx.shared.DICT.flush\_all

语法：

```
ngx.shared.DICT:flush_all()
```

上下文：init\_by\_lua\*、set\_by\_lua\*、rewrite\_by\_lua\*、access\_by\_lua\*、content\_by\_lua\*、header\_filter\_by\_lua\*、body\_filter\_by\_lua\*、log\_by\_lua\*、ngx.timer.\*、balancer\_by\_lua\*、ssl\_certificate\_by\_lua\*、ssl\_session\_fetch\_by\_lua\*、ssl\_session\_store\_by\_lua\*。

说明：清除字典中的所有元素。本方法不实际释放所有的内存块，只是把所有的元素标记为过期。

### 93. ngx.shared.DICT.flush\_expired

语法：

```
flushed = ngx.shared.DICT:flush_expired(max_count?)
```

上下文：init\_by\_lua\*、set\_by\_lua\*、rewrite\_by\_lua\*、access\_by\_lua\*、content\_by\_lua\*、header\_filter\_by\_lua\*、body\_filter\_by\_lua\*、log\_by\_lua\*、ngx.timer.\*、balancer\_by\_lua\*、ssl\_certificate\_by\_lua\*、ssl\_session\_fetch\_by\_lua\*、ssl\_session\_store\_by\_lua\*。

说明：清除所有过期的元素，max\_count 表明清除的上限数量。当 max\_count 为 0 或没有指定的时候，意味着没有限制。返回值是实际清除掉的元素数目。和 flush\_all 方法不同，这个方法会实际释放被过期元素占用的内存。

### 94. ngx.shared.DICT.get\_keys

语法：

```
keys = ngx.shared.DICT:get_keys(max_count?)
```

上下文：init\_by\_lua\*、set\_by\_lua\*、rewrite\_by\_lua\*、access\_by\_lua\*、content\_by\_lua\*、

header\_filter\_by\_lua\*、body\_filter\_by\_lua\*、log\_by\_lua\*、ngx.timer.\*、balancer\_by\_lua\*、ssl\_certificate\_by\_lua\*、ssl\_session\_fetch\_by\_lua\*、ssl\_session\_store\_by\_lua\*。

说明：从字典中最多获取 max\_count 条 key，返回一个 key 的列表。默认地，只会返回前 1024 个 key。当 <max\_count> 参数给定为 0 时，所有的元素都会被返回。

注意，在字典中有大量 key 的情况下要慎重使用本方法。这个方法将锁定字典很长一段时间，所有其他工作进程都将因为字典被占用而不断重试，导致处理速度降低。

## 95. ngx.socket.udp

语法：

```
udpsock = ngx.socket.udp()
```

上下文：rewrite\_by\_lua\*、access\_by\_lua\*、content\_by\_lua\*、ngx.timer.\*、ssl\_certificate\_by\_lua\*、ssl\_session\_fetch\_by\_lua\*。

说明：创建并返回一个 UDP 套接字或数据报基础的 UNIX 域套接字（cosocket 对象）。可以在对象上使用下面的方法：setpeername、send、receive、close、settimeout。

本对象计划兼容 LuaSocket 的 UDP API，但是是 100% 非阻塞的。

## 96. udpsock:setpeername

语法：

```
ok, err = udpsock:setpeername(host, port)
ok, err = udpsock:setpeername("unix:/path/to/unix-domain.socket")
```

上下文：rewrite\_by\_lua\*、access\_by\_lua\*、content\_by\_lua\*、ngx.timer.\*、ssl\_certificate\_by\_lua\*、ssl\_session\_fetch\_by\_lua\*。

说明：准备连接一个套接字对象到一个远程服务上的 UDP 服务或 UNIX 域套接字文件。因为数据报是无连接的，这个过程没有实际建立一个连接，只是为子请求的读和写设置了远程服务的名字。

可以在 host 参数上使用 IP 地址和域名。如果使用域名，方法将调用 Nginx 核心的非阻塞动态域名解析器，但这个需要在 nginx.conf 文件配置 resolver 指令：

```
resolver8.8.8.8; # use Google's public DNS nameserver
```

如果名字服务返回多个 IP 地址，方法将随机选取一个。发生任何错误，返回 nil，err 包含错误描述。如果成功了，ok 返回的是 1。这是一个连接到 UDP 服务器的例子：

```
location/test {
    resolver8.8.8.8;

    content_by_lua_block {
        local sock = ngx.socket.udp()
        local ok, err = sock:setpeername("my.memcached.server.domain", 11211)
        if not ok then
            ngx.say("failed to connect to memcached: ", err)
            return
        end
    }
}
```

```

        end
        ngx.say("successfully connected to memcached!")
        sock:close()
    }
}

```

从 v0.7.18 版本开始, 在 Linux 上也可以连接到数据报的域套接字:

```

local sock = ngx.socket.udp()
local ok, err = sock:setpeername("unix:/tmp/some-datagram-service.sock")
if not ok then
    ngx.say("failed to connect to the datagram unix domain socket: ", err)
    return
end

```

假设数据报服务监听在 UNIX 域套接字文件 /tmp/some-datagram-service.sock, 客户端套接字要使用 Linux 的 autobind 性能。

在一个已经连接的套接字对象上调用这个方法, 将使原来的连接先关闭。

### 97. udpsock:send

语法:

```
ok, err = udpsock:send(data)
```

上下文: rewrite\_by\_lua\*、access\_by\_lua\*、content\_by\_lua\*、ngx.timer.\*、ssl\_certificate\_by\_lua\*、ssl\_session\_fetch\_by\_lua\*。

说明: 在当前 UDP 或 UNIX 域套接字上发送数据。发送成功, 返回 1, 否则返回 nil 和一个错误描述字符串。data 可以是一个 Lua 字符串或一个由字符串元素构成的 Lua 表 (可以是嵌套的)。在表的情况下, 方法将复制所有的字符串元素到底层 Nginx 套接字发送缓冲区, 这比在 Lua 端进行字符串拼接更有效率。

### 98. udpsock:receive

语法:

```
data, err = udpsock:receive(size?)
```

上下文: rewrite\_by\_lua\*、access\_by\_lua\*、content\_by\_lua\*、ngx.timer.\*、ssl\_certificate\_by\_lua\*、ssl\_session\_fetch\_by\_lua\*。

说明: 从 UDP 或 UNIX 域套接字对象接收 size 数量的数据。这个方法是一个同步操作, 但是是 100% 非阻塞的。读取成功了, 返回接收到的数据; 发生任何一种错误, 将返回 nil 和一个错误描述字符串。如果指定了 size 参数, 那么将使用 size 作为接收缓冲区尺寸。如果 size 大于 8192, 那么将使用 8192 代替。如果没有参数指定, 将使用 8192 作为最大的尺寸。读取的超时值由 lua\_socket\_read\_timeout 指令控制或者通过 settimeout 方法设置, settimeout 值优先。例如:

```

sock:settimeout(1000) -- one second timeout
local data, err = sock:receive()

```

```

ifnot data then
    ngx.say("failed to read a packet: ", err)
    return
end
ngx.say("successfully read a packet: ", data)

```

在 receive 之前调用 settimeout 是非常重要的。

## 99. udpsock:close

语法:

```
ok, err = udpsock:close()
```

上下文: rewrite\_by\_lua\*、access\_by\_lua\*、content\_by\_lua\*、ngx.timer.\*、ssl\_certificate\_by\_lua\*、ssl\_session\_fetch\_by\_lua\*。

说明: 关闭当前 UDP 或 UNIX 域套接字。ok 返回 1 表示成功, ok 返回 nil 和 err 错误描述信息表示失败。当套接字对象被 Lua GC (垃圾回收器) 回收或当前客户端 HTTP 请求结束处理时, 尽管没有调用过本方法, 连接也会被关闭。

## 100. udpsock:settimeout

语法:

```
udpsock:settimeout(time)
```

上下文: rewrite\_by\_lua\*、access\_by\_lua\*、content\_by\_lua\*、ngx.timer.\*、ssl\_certificate\_by\_lua\*、ssl\_session\_fetch\_by\_lua\*。

说明: 设置子请求套接字操作的超时值, 单位为毫秒。本操作设置的值优先级比 lua\_socket\_read\_timeout 指令高。

## 101. ngx.socket.stream

说明: ngx.socket.tcp 的一个别名。如果流式的 cosocket 可以连接到一个 UNIX 域套接字上, 那么这个 API 名字是首选的。

## 102. ngx.socket.tcp

语法:

```
tcpsock = ngx.socket.tcp()
```

上下文: rewrite\_by\_lua\*、access\_by\_lua\*、content\_by\_lua\*、ngx.timer.\*、ssl\_certificate\_by\_lua\*、ssl\_session\_fetch\_by\_lua\*。

说明: 创建并且返回一个 TCP 或流式 UNIX 域套接字对象 (cosocket 对象)。对象提供下面的方法: connect、sslhandshake、send、receive、close、settimeout、settimeouts、setoption、receiveuntil、setkeepalive、getreusedtimes。

本对象考虑兼容 LuaSocket 库的 TCP API, 但是是 100% 非阻塞的。这个 API 创建的 cosocket 对象和创建它的 Lua 处理器拥有相同的生命周期。所以, 永远不要把 cosocket 传

递到其他的处理器中（包括 ngx.timer 的回调函数），也永远不要在不同的 Nginx 请求中共享 cosocket 对象。

对于每一个 cosocket 对象的底层连接，如果不能明确地关闭它（通过 close）或放到连接池的后面（通过 setkealive），那么发生下面两种事件时会被自动关闭：

- 当前请求处理器工作完成；
- cosocket 对象值被 Lua GC 回收了。

cosocket 操作上的致命错误总是会关闭当前连接（读超时错误只是一个不致命的错误），如果在一个已经关闭的连接上调用 close，会得到 “closed” 错误。

从 0.9.9 版本开始，cosocket 对象是全双工的，意味着在一个 cosocket 对象上读和写可以分别由一个协程同时处理（两个协程必须同属于相同的处理器）。但是不能有两个协程同时读或同时写或同时连接同一个 cosocket 对象，否则，会得到 “socket busy reading” 的错误。

### 103. tcpsock:connect

语法：

```
ok, err = tcpsock:connect(host, port, options_table?)
ok, err = tcpsock:connect("unix:/path/to/unix-domain.socket", options_table?)
```

上下文: rewrite\_by\_lua\*、access\_by\_lua\*、content\_by\_lua\*、ngx.timer.\*、ssl\_certificate\_by\_lua\*、ssl\_session\_fetch\_by\_lua\*。

说明：非阻塞式地连接一个远程的 TCP 服务或一个流式的 UNIX 域套接字。实际解析主机名并且连接到远程后端之前，这个方法总是遍历连接池，查找之前，这个方法调用创建的匹配的空闲连接（或者 ngx.socket.connect 函数）。

IP 地址和域名都可以指定为 host 参数。如果 host 是域名，方法将使用 Nginx 核心非阻塞的动态解析器。解析器通过 resolver 指令配置在 nginx.conf 中，例如：

```
resolver8.8.8.8; # use Google's public DNS nameserver
```

如果返回多个 IP，方法会随机选择一个。发生任何错误时，ok 会返回 nil，err 会返回一个错误描述字符串。ok 为 1 则表示成功。

下面是一个连接 TCP 服务的例子：

```
location/test {
    resolver8.8.8.8;

    content_by_lua_block {
        local sock = ngx.socket.tcp()
        local ok, err = sock:connect("www.google.com", 80)
        if not ok then
            ngx.say("failed to connect to google: ", err)
            return
        end
        ngx.say("successfully connected to google!")
        sock:close()
    }
}
```

```
}
}
```

也可能连接到 UNIX 域套接字:

```
local sock = ngx.socket.tcp()
local ok, err = sock:connect("unix:/tmp/memcached.sock")
if not ok then
    ngx.say("failed to connect to the memcached unix domain socket: ", err)
    return
end
```

假设 memcached ( 或其他的 ) 监听在 /tmp/memcached.sock 这个 UNIX 域套接字文件上。

超时值由 lua\_socket\_connect\_timeout 指令控制, 也可以通过 settimeout 方法设置, 而且方法的数值优先, 例如:

```
local sock = ngx.socket.tcp()
sock:settimeout(1000) -- one second timeout
local ok, err = sock:connect(host, port)
```

在本方法之前调用 settimeout 方法是很重要的。在一个已经连接成功的套接字对象上调用本方法, 将会导致原始旧的套接字首先被关闭。

可以使用一个 Lua 表作为可选的最后一个参数, 以指定多种连接参数: pool 为用到的连接池指定了一个自定义名字。如果未指定, 那么连接池名字会使用模板 “<host>:<port>” 或 “<unix-socket-path>”。

#### 104. tcpsock:sslhandshake

语法:

```
session, err = tcpsock:sslhandshake(reused_session?, server_name?, ssl_verify?,
    send_status_req?)
```

上下文: rewrite\_by\_lua\*、access\_by\_lua\*、content\_by\_lua\*、ngx.timer.\*、ssl\_certificate\_by\_lua\*、ssl\_session\_fetch\_by\_lua\*。

说明: 在当前建立起的连接上做 SSL/TLS 握手。

可选参数 reused\_session 可以携带一个由之前相同目标的握手方法产生的最近的 SSL 会话自定义数据 (ID)。对于短生命周期连接, 重用 SSL 会话可以将握手速度提高一个数量级。但是, 如果连接池是使能状态, 则重用就不是很有效了。此参数默认是 nil。如果此参数是 false, 将没有 SSL 会话用户数据返回, 而且只有一个 Lua 布尔值作为第一个返回值; 否则, 成功的情况下, 当前 SSL 会话将总是作为第一个参数返回。

可选参数 server\_name 用于指定新的 TLS 扩展服务名字 (SNI)。使用 SNI 可以使不同的服务共享相同的 IP 地址。同样, 当 SSL 校验打开的时候, server\_name 也总是用于校验远程服务发来的服务器证书中的名字。

可选参数 ssl\_verify 用一个 Lua 布尔值控制是否执行 SSL 验证。当该参数为 true 时, 服务器证书将通过 CA 中心验证, CA 中心通过 lua\_ssl\_trusted\_certificate 指令指定。可能

总是需要调节 `lua_ssl_verify_depth` 以控制需要的校验链深度。同样，当 `ssl_verify` 为 `true`，且 `server_name` 参数也指定了时，可使用 `server_name` 机制验证证书中的服务名。

可选参数 `send_status_req` 用一个布尔值控制 SSL 握手是否发送 OCSP 状态请求，在一个已经创建好 SSL/TLS 握手的连接上，本方法会立即返回。

### 105. tcpsock:send

语法：

```
bytes, err = tcpsock:send(data)
```

上下文: `rewrite_by_lua*`、`access_by_lua*`、`content_by_lua*`、`ngx.timer.*`、`ssl_certificate_by_lua*`、`ssl_session_fetch_by_lua*`。

说明：非阻塞地在当前 TCP 或 UNIX 域套接字连接上发送数据。

本方法是一个同步操作，除非所有的数据被写到系统套接字发送缓冲区内或有一个错误发生，否则不会返回。成功的情况下，`bytes` 返回已经发送的字节数；否则，`bytes` 返回 `nil`，`err` 返回一个错误描述信息。

`data` 可以是一个 Lua 字符串或一个由字符串元素构成的 Lua 表（可以是嵌套的）。在 `data` 为表的情况下，方法将复制所有的字符串元素到底层 Nginx 套接字发送缓冲区，这比在 Lua 端进行字符串拼接更有效率。

发送超时值通过 `lua_socket_send_timeout` 指令控制，或者通过 `settimeout` 方法，后者优先级更高。例如：

```
sock:settimeout(1000) -- one second timeout
local bytes, err = sock:send(request)
```

调用本方法之前调用 `settimeout` 方法是非常重要的。

### 106. tcpsock:receive

语法：

```
data, err, partial = tcpsock:receive(size)
data, err, partial = tcpsock:receive(pattern?)
```

上下文: `rewrite_by_lua*`、`access_by_lua*`、`content_by_lua*`、`ngx.timer.*`、`ssl_certificate_by_lua*`、`ssl_session_fetch_by_lua*`。

说明：依据读取模式或 `size` 从当前连接上接收数据。

本方法和 `send` 方法一样，是一个 100% 不阻塞的同步操作。接收成功，`data` 返回收到的数据。任何错误情况下，`data` 为 `nil`，`err` 返回描述错误的字符串，`partial` 返回错误发生时收到的部分数据。如果指定了类数据型的 `size` 参数，将被作为一个接收的尺寸。在收到指定大小数据或发生错误之前不会返回。

如果指定了一个非数值型字符串，那么将当作一个模板使用。支持下列模板：

- `'*a'`：在套接字读取直到连接关闭，不执行行尾转换。



- `'*l'`：读取一行文本，一行是用换行符（LF，ASCII 10）标记的，可选的，回车符（CR，ASCII 13）也表示一行结束。LF 和 CR 不包含在返回的行数据中。事实上，所有的 CR 都会被模板忽略。

如果没有指定参数，系统默认使用 `'*l'` 模板，默认读取一行。

读取操作的超时值由 `lua_socket_read_timeout` 指令控制，也可以通过 `settimeout` 方法操作，`settimeout` 方法优先。下面是一个例子：

```
sock:settimeout(1000) -- one second timeout
local line, err, partial = sock:receive()
if not line then
    ngx.say("failed to read a line: ", err)
    return
end
ngx.say("successfully read a line: ", line)
```

调用本方法之前调用 `settimeout` 方法是非常重要的。

从 v0.8.8 版本开始，当读取超时错误发生时，不会自动关闭当前连接。在其他的方法中，`receive` 方法会自动在错误发生的情况下关闭连接。

### 107. `tcpsock:receiveuntil`

语法：

```
iterator = tcpsock:receiveuntil(pattern, options?)
```

上下文：`rewrite_by_lua*`、`access_by_lua*`、`content_by_lua*`、`ngx.timer.*`、`ssl_certificate_by_lua*`、`ssl_session_fetch_by_lua*`。

说明：返回一个 Lua 迭代器函数，可以用来读取数据值，直到指定的模板出现或发生错误。例如：

```
local reader = sock:receiveuntil("\r\n--abcdh")
local data, err, partial = reader()
if not data then
    ngx.say("failed to read the data stream: ", err)
end
ngx.say("read the data stream: ", data)
```

当不使用任何参数调用时，迭代器函数返回指定的模板字符串之前的数据。

当错误发生时，迭代器函数返回 `nil` 和错误信息描述字符串，`partial` 中存放错误发生时已经读到的数据。

迭代器函数可以多次调用，并且可以和其他 `cosocket` 方法混合使用。

当使用一个 `size` 参数调用迭代器函数时，它的行为将非常不同。因为它在每次读取时只读取 `size` 大小的数据，在最后一次调用时返回 `nil`（依模板界线或遇到了错误）。最后一次成功调用时，`err` 返回的值也是 `nil`。最后一次成功调用后，迭代器函数将被复位，返回 `nil` 的 `data` 值和 `nil` 的 `err` 值。考虑下面的例子：



```

local reader = sock:receiveuntil("\r\n--abcedhb")

while true do
    local data, err, partial = reader(4)
    if not data then
        if err then
            ngx.say("failed to read the data stream: ", err)
            break
        end

        ngx.say("read done")
        break
    end

    ngx.say("read chunk: [", data, "]")
end

```

输入数据是 'hello, world! -agentzh\r\n--abcedhb blah blah' 的情况下, 我们将得到下面的输出:

```

read chunk: [hell]
read chunk: [o, w]
read chunk: [orld]
read chunk: [! -a]
read chunk: [gent]
read chunk: [zh]
read done

```

注意, 当模板在解析流时有不明确的时候, 实际收到的数据可能比 size 要求的长一点。同样, 也可能是 data 返回的数据稍短一点点。

读取操作的超时值由 lua\_socket\_read\_timeout 指令控制, 也可以通过 settimeout 方法操作, settimeout 方法优先。例如:

```

local readline = sock:receiveuntil("\r\n")

sock:settimeout(1000) -- one second timeout
line, err, partial = readline()
if not line then
    ngx.say("failed to read a line: ", err)
return
end
ngx.say("successfully read a line: ", line)

```

在迭代函数之前调用 settimeout 是非常重要的, 但是 receiveuntil 函数用不到这个值, 所以是不相关的。

从 v0.5.1 版本开始, receiveuntil 支持可选的 option 选项, 并支持下面的选项:

- inclusive 是布尔型值, 控制是否在返回的数据串中包含模板串, 默认是 false。例如:

```

local reader = tcpsock:receiveuntil("_END_", { inclusive = true })
local data = reader()
ngx.say(data)

```

对于“hello world \_END\_ blah blah blah”这串数据，上面的例子将输出“hello world \_END\_”，包含了模板字符串“\_END\_”。

从 v0.8.8 版本开始，当读取超时错误发生时，不会自动关闭当前连接。在其他的方法中，本方法会自动在错误发生的情况下关闭连接。

#### 108. tcpsock:close

语法：

```
ok, err = tcpsock:close()
```

上下文: `rewrite_by_lua*`、`access_by_lua*`、`content_by_lua*`、`ngx.timer.*`、`ssl_certificate_by_lua*`、`ssl_session_fetch_by_lua*`。

说明：关闭当前 TCP 或 UNIX 域套接字。成功时 `ok` 返回 1，失败时 `ok` 返回 `nil`，`err` 存放错误描述字符串。

注意，当连接已经调用 `setkeepalive` 方法后，不需要再调用 `close`，因为套接字会自动关闭（并且当前连接保存在内建的连接池中），套接字对象不需要调用本方法。当套接字对象被 Lua GC 释放或 HTTP 处理过程结束的时候，将被自动关闭，不需要手工再次调用。

#### 109. tcpsock:settimeout

语法：

```
tcpsock:settimeout(time)
```

上下文: `rewrite_by_lua*`、`access_by_lua*`、`content_by_lua*`、`ngx.timer.*`、`ssl_certificate_by_lua*`、`ssl_session_fetch_by_lua*`。

说明：设置套接字的超时值，单位为毫秒。受影响的方法为 `connect`、`receive` 和 `receiveuntil` 返回的迭代函数。本方法设置的值比指令中的数据优先级高，如 `lua_socket_connect_timeout`、`lua_socket_send_timeout`、`lua_socket_read_timeout`。

注意，本方法不影响 `lua_socket_keepalive_timeout` 设置。可使用 `setkeepalive` 方法修改其设置。

#### 110. tcpsock:settimeouts

语法：

```
tcpsock:settimeouts(connect_timeout, send_timeout, read_timeout)
```

上下文: `rewrite_by_lua*`、`access_by_lua*`、`content_by_lua*`、`ngx.timer.*`、`ssl_certificate_by_lua*`、`ssl_session_fetch_by_lua*`。

说明：分别设置连接、发送和读操作的超时值，单位为毫秒，同时也影响 `receiveuntil` 返回的迭代函数。本方法优先级高于 `lua_socket_connect_timeout`、`lua_socket_send_timeout` 和 `lua_socket_read_timeout` 指令。推荐使用 `settimeouts` 代替 `settimeout`。

注意，本方法不影响 `lua_socket_keepalive_timeout` 设置。可使用 `setkeepalive` 方法修改

其设置。

### 111. tcpsock:setoption

语法:

```
tcpsock:setoption(option, value?)
```

上下文: `rewrite_by_lua*`、`access_by_lua*`、`content_by_lua*`、`ngx.timer.*`、`ssl_certificate_by_lua*`、`ssl_session_fetch_by_lua*`。

说明: 本函数是为了兼容 LuaSocket API 加进来的, 但是当前不做任何事情, 未来会实现。

### 112. tcpsock:setkeepalive

语法:

```
ok, err = tcpsock:setkeepalive(timeout?, size?)
```

上下文: `rewrite_by_lua*`、`access_by_lua*`、`content_by_lua*`、`ngx.timer.*`、`ssl_certificate_by_lua*`、`ssl_session_fetch_by_lua*`。

说明: 把当前的连接立即放进 cosocket 内建的连接池, 并且保活, 直到其他的 connect 方法请求它或者关联的最大空闲时间到期。

可选的 timeout 参数用来指定最大空闲超时时间, 单位为毫秒。如果没有指定 timeout 参数, 使用 `lua_socket_keepalive_timeout` 指令的值。如果指定 timeout 为 0, 则表示永远不会过期。

第二个可选参数 size 用来指定连接池允许的最大数量。注意, 连接池的容量在创建后也可以修改。当未指定这个参数时, 使用 `lua_socket_pool_size` 指令。

当连接池超过容量限制时, 将根据 LRU 算法将空闲时间最久的连接关闭, 为当前连接释放空间。

注意, cosocket 连接池是工作进程级别的, 而不是 Nginx 服务级别的, 所以在这里设置的连接池 size 也会被应用到每一个工作进程中的。

池中的空闲连接会被监控, 每一个异常事情, 如断开、非预期的输入数据等, 会导致连接被关闭并从池中移除。

成功时, 方法返回 1, 失败时则返回 nil 和一个错误描述字符串。

当前连接的系统接收缓冲区中有未读数据时, 将返回 “connection in dubious state” 错误信息, 因为之前的会话未把数据读取完全, 有数据遗留, 这种情况对于后续连接是不安全的。

本方法总是使当前 cosocket 对象进入 “closed” 状态, 所以不需要手工调用 close 方法。

### 113. tcpsock:getreusedtimes

语法:

```
count, err = tcpsock:getreusedtimes()
```

上下文: `rewrite_by_lua*`、`access_by_lua*`、`content_by_lua*`、`ngx.timer.*`、`ssl_certificate_by_lua*`、`ssl_session_fetch_by_lua*`。

说明: 返回当前连接重用次数。如果失败, 则返回 `nil` 和错误描述字符串。

如果当前连接不是来自于内建的连接池, 将返回 0, 表示这个连接未被重用过。如果本连接来自于连接池, 则返回值是非 0 值。

#### 114. ngx.socket.connect

语法:

```
tcpsock, err = ngx.socket.connect(host, port)
tcpsock, err = ngx.socket.connect("unix:/path/to/unix-domain.socket")
```

上下文: `rewrite_by_lua*`、`access_by_lua*`、`content_by_lua*`、`ngx.timer.*`。

说明: 本方法把 `ngx.socket.tcp()` 和 `connect()` 组合成单一操作, 下面是一个实际应用例子。

```
local sock = ngx.socket.tcp()
local ok, err = sock:connect(...)
if not ok then
    return nil, err
end
return sock
```

`settimeout` 无法为本方法指定连接超时值, 需要使用 `lua_socket_connect_timeout` 指令。

#### 115. ngx.get\_phase

语法:

```
str = ngx.get_phase()
```

上下文: `init_by_lua*`、`init_worker_by_lua*`、`set_by_lua*`、`rewrite_by_lua*`、`access_by_lua*`、`content_by_lua*`、`header_filter_by_lua*`、`body_filter_by_lua*`、`log_by_lua*`、`ngx.timer.*`、`balancer_by_lua*`、`ssl_certificate_by_lua*`、`ssl_session_fetch_by_lua*`、`ssl_session_store_by_lua*`。

说明: 获取当前运行阶段名字, 可能的返回值如下。

- `init` for the context of `init_by_lua*`;
- `init_worker` for the context of `init_worker_by_lua*`;
- `ssl_cert` for the context of `ssl_certificate_by_lua*`;
- `ssl_session_fetch` for the context of `ssl_session_fetch_by_lua*`;
- `ssl_session_store` for the context of `ssl_session_store_by_lua*`;
- `set` for the context of `set_by_lua*`;
- `rewrite` for the context of `rewrite_by_lua*`;
- `balancer` for the context of `balancer_by_lua*`;
- `access` for the context of `access_by_lua*`;
- `content` for the context of `content_by_lua*`;

- header\_filter for the context of header\_filter\_by\_lua\*;
- body\_filter for the context of body\_filter\_by\_lua\*;
- log for the context of log\_by\_lua\*;
- timer for the context of user callback functions for ngx.timer.\*。

## 116. ngx.thread.spawn

语法:

```
co = ngx.thread.spawn(func, arg1, arg2, ...)
```

上下文: rewrite\_by\_lua\*、access\_by\_lua\*、content\_by\_lua\*、ngx.timer.\*、ssl\_certificate\_by\_lua\*、ssl\_session\_fetch\_by\_lua\*。

说明: 使用 func 为运行函数创建一个新的用户级轻量化线程, 参数为 arg1、arg2 等。返回值 co 是 Lua 线程 (或 Lua 协程) 代表这个轻量化线程的协程对象。轻量化线程是一种 ngx\_lua 模块调度的特殊的 Lua 协程。

ngx.thread.spawn 返回之前, func 函数会被使用参数 arg1、arg2 等调用, 直到返回。任何错误都会导致函数返回, 或者因为通过 Nginx Lua API 进行 I/O 操作 (如 tcpsocket.receive) 导致挂起。ngx.thread.spawn 返回以后, 新创建的轻量化线程将在各种 I/O 事件中保持异步运行。所有 rewrite\_by\_lua、access\_by\_lua、content\_by\_lua 运行起来的 Lua 代码块, 都会被 ngx\_lua 自动创建一个样板协程来运行。这个样板轻线程名字为 “entry threads”。

默认地, 相关的 Nginx 处理器 (如 rewrite\_by\_lua 处理器) 直到遇到下列情况, 否则不会中止:

- 所有的 “entry thread” 和所有的用户轻量化线程中止。
- “entry thread” 或一个用户轻量化线程通过调用 ngx.exit、ngx.exec、ngx.redirect、ngx.req.set\_uri(uri, true) 中止。
- “entry thread” 遇到一个 Lua 错误中止。

当轻线程遇到一个 Lua 错误中止的时候, 不会终止其他轻线程。因为 Nginx 子请求模式的限制, 常规上不允许中断一个正在运行的子请求, 也不允许中断待定状态的线程, 必须使用 ngx.thread.wait 等待协程中止。在严重异常的情况下, 可以使用 ngx.ERROR(-1)、408、444 或 499 状态码的 ngx.exit 中断待定的子请求。

轻线程不是按抢先模式调度的, 所以不会被自动分配时间片。线程排外地在 CPU 上运行, 直到:

- 一个 (非阻塞) I/O 操作不能够在一次单一的运行中完成。
- 通过 coroutine.yield 放弃运行。
- Lua 错误中断或调用了 ngx.exit、ngx.exec、ngx.redirect 或 ngx.req.set\_uri(uri, true)。

前两种情况下, 纯线程通常会被 ngx\_lua 调度器恢复, 除非整个进程退出了。

轻线程可以创建自己的轻线程。一个通过 `coroutine.create` 创建的协程也可以创建“轻线程”。一个直接创建轻线程的协程（可以是普通的 Lua 协程或轻线程）叫做新线程的“父协程”。

父协程可以通过调用 `ngx.thread.wait` 等待所有的子轻线程退出。可以在轻线程的协程上调用 `coroutines.status()` 和 `coroutine.yield()`。

线程对应的协程状态可以是迟钝的状态，如果：

- 当前线程已经中断（成功或发生错误）。
- 父协程仍然存在。
- 父协程没有在 `ngx.thread.wait` 上等待。

下面的例子演示在轻线程的协程对象上通过 `coroutine.yield()` 手工控制时间片。

```
local yield = coroutine.yield

function f()
    local self=coroutine.running()

    ngx.say("f 1")
    yield(self)

    ngx.say("f 2")
    yield(self)

    ngx.say("f 3")
end

local self=coroutine.running()
ngx.say("0")
yield(self)

ngx.say("1")
ngx.thread.spawn(f)

ngx.say("2")
yield(self)

ngx.say("3")
yield(self)

ngx.say("4")
```

会得到输出：

```
0
1
f 1
2
f 2
3
f 3
4
```

轻线程在单个 Nginx 请求处理器中同时并发处理上游请求时非常有用，有点像 ngx.location.capture\_multi 可以和所有的 Nginx Lua API 工作一样。下面的例子演示在单 Lua 处理器中并发的 MySQL、Memcached 和上游 HTTP 服务请求，并且输出按照实际返回顺序的结果（与 Facebook 的 BigPipe 模式很像）：

```
-- query mysql, memcached, and a remote http service at the same time,
-- output the results in the order that they
-- actually return the results.
```

```
local mysql = require "resty.mysql"
local memcached = require "resty.memcached"
```

```
localfunction query_mysql()
local db = mysql:new()
    db:connect{
        host = "127.0.0.1",
        port = 3306,
        database = "test",
        user = "monty",
        password = "mypass"
    }
end
```

```
local res, err, errno, sqlstate =
    db:query("select * from cats order by id asc")
    db:set_keepalive(0, 100)
    ngx.say("mysql done: ", cJSON.encode(res))
end
```

```
localfunction query_memcached()
local memc = memcached:new()
    memc:connect("127.0.0.1", 11211)
    local res, err = memc:get("some_key")
    ngx.say("memcached done: ", res)
end
```

```
localfunction query_http()
local res = ngx.location.capture("/my-http-proxy")
    ngx.say("http done: ", res.body)
end
```

```
ngx.thread.spawn(query_mysql)      -- create thread 1
ngx.thread.spawn(query_memcached) -- create thread 2
ngx.thread.spawn(query_http)      -- create thread 3
```

## 117. ngx.thread.wait

语法：

```
ok, res1, res2, ... = ngx.thread.wait(thread1, thread2, ...)
```

上下文: rewrite\_by\_lua\*、access\_by\_lua\*、content\_by\_lua\*、ngx.timer.\*、ssl\_certificate\_by\_lua\*、ssl\_session\_fetch\_by\_lua\*。

说明：等待一个或多个轻线程，返回第一个中断线程的状态，无论成功或失败。

thread1、thread2 是使用 ngx.thread.spawn 创建的线程对象（协程）。返回值返回和 coroutine.resume 拥有完全相同的意思，ok 值是一个布尔型值，指出线程是否退出，res1、res2 用于存放协程函数返回值或错误的对象（失败的情况）。

只有直接的父协程才能够等待它的子线程，否则会抛出一个 Lua 异常。下面的例子演示 ngx.thread.wait 和 ngx.location.capture 模拟 ngx.location.capture\_multi：

```
local capture = ngx.location.capture
local spawn = ngx.thread.spawn
local wait = ngx.thread.wait
local say = ngx.say

localfunction fetch(uri)
    return capture(uri)
end

local threads = {
    spawn(fetch, "/foo"),
    spawn(fetch, "/bar"),
    spawn(fetch, "/baz")
}

for i =1, #threads do
    local ok, res = wait(threads[i])
    if not ok then
        say(i, ": failed to run: ", res)
    else
        say(i, ": status: ", res.status)
        say(i, ": body: ", res.body)
    end
end
```

这是一个典型的 wait all 模式。下面的例子演示 wait any 模式：

```
function f()
    ngx.sleep(0.2)
    ngx.say("f: hello")
    return "f done"
end

function g()
    ngx.sleep(0.1)
    ngx.say("g: hello")
    return "g done"
end

local tf, err = ngx.thread.spawn(f)
if not tf then
    ngx.say("failed to spawn thread f: ", err)
    return
end

ngx.say("f thread created: ", coroutine.status(tf))
```



```

local tg, err = ngx.thread.spawn(g)
if not tg then
    ngx.say("failed to spawn thread g: ", err)
    return
end

ngx.say("g thread created: ", coroutine.status(tg))

ok, res = ngx.thread.wait(tf, tg)
if not ok then
    ngx.say("failed to wait: ", res)
    return
end

ngx.say("res: ", res)

```

```

-- stop the "world", aborting other running threads
ngx.exit(ngx.OK)

```

输出:

```

f thread created: running
g thread created: running
g: hello
res: g done

```

## 118. ngx.thread.kill

语法:

```
ok, err = ngx.thread.kill(thread)
```

上下文: `rewrite_by_lua*`、`access_by_lua*`、`content_by_lua*`、`ngx.timer.*`。

说明: 杀掉一个通过 `ngx.thread.spawn` 创建的轻线程。如果成功, `ok` 返回 `true` 值, 否则, `ok` 返回 `nil`, `err` 返回错误描述。

本函数依赖于当前实现。只有父协程 (或轻线程) 可以杀掉一个线程, 因为 Nginx 核心的限制, 不能杀掉一个包含 Nginx 子请求处于待定状态的线程 (如通过 `ngx.location.capture` 初始化的)。

## 119. ngx.on\_abort

语法:

```
ok, err = ngx.on_abort(callback)
```

上下文: `rewrite_by_lua*`、`access_by_lua*`、`content_by_lua*`。

说明: 把一个 Lua 函数作为回调函数注册, 当用户端 (下游) 过早地关闭连接时, 会被自动调用。如果注册成功, `ok` 返回 1, 否则返回 `nil`, `err` 返回错误提示信息。

所有的 Nginx Lua API 都可以在回调函数中使用, 因为函数运行在一个特殊的轻线程中, 就像这些线程是用 `ngx.thread.spawn` 创建出来的一样。回调函数可以在客户端中断时决

定做些什么。例如，可以什么也不做，这样当前请求处理器将继续运行。回调函数也可以通过 `ngx.exit` 中断处理，例如：

```
localfunction my_cleanup()
-- custom cleanup work goes here, like cancelling a pending DB transaction

-- now abort all the "light threads" running in the current request handler
    ngx.exit(499)
end

local ok, err = ngx.on_abort(my_cleanup)
ifnot ok then
    ngx.log(ngx.ERR, "failed to register the on_abort callback: ", err)
    ngx.exit(500)
end
```

`lua_check_client_abort` 指令设置为 `off` (默认值) 时，调用本函数时将总是返回 “`lua_check_client_abort is off`” 错误信息。

本函数依赖于当前实现。在一个请求处理器中，本回调只会被调用一次，子请求再调本函数会得到错误信息 “`duplicate call`”。

## 120. ngx.timer.at

语法：

```
ok, err = ngx.timer.at(delay, callback, user_arg1, user_arg2, ...)
```

上下文: `init_worker_by_lua*`、`set_by_lua*`、`rewrite_by_lua*`、`access_by_lua*`、`content_by_lua*`、`header_filter_by_lua*`、`body_filter_by_lua*`、`log_by_lua*`、`ngx.timer.*`、`balancer_by_lua*`、`ssl_certificate_by_lua*`、`ssl_session_fetch_by_lua*`、`ssl_session_store_by_lua*`。

说明：使用可选的用户参数和 `callback` 回调函数创建一个用户自定义时钟。

`delay` 指定时钟从什么时候开始，可以使用小数如 `0.001` 表示 1 毫秒。`0` 表示立即过期（马上回调函数被调用）。第二个参数是 Lua 函数，在时钟到期时会在一个轻线程中被调用。

`user_arg1`、`user_arg2` 等参数是 `callback` 的参数，函数被调用时会传入这些参数。

当 Nginx 工作进程尝试关闭时，时钟会提前到期，函数提前被调用。例如，Nginx 收到 HUP 信号，将重新载入配置文件。这时调用 `ngx.timer.at` 创建非零延迟的新时钟，将返回 `nil` 和 “`process exiting`” 错误消息。

从 v0.9.3 版本开始，在工作进程开始关闭的时候允许创建零延迟的时钟。

当一个时钟到期了，回调函数中的 Lua 代码运行在一个从创建时钟的请求中分离的轻线程中。所以，与其相同生命周期的请求创建的对象，如 `cosockets`，不能够在原始请求和时钟回调函数中共享。

这是一个简单的例子：

```
location/ {
```

```
...
```

```

log_by_lua_block {
    local function push_data(premature, uri, args, status)
        -- push the data uri, args, and status to the remote
        -- via ngx.socket.tcp or ngx.socket.udp
        -- (one may want to buffer the data in Lua a bit to
        -- save I/O operations)
    end
    local ok, err = ngx.timer.at(0, push_data, ngx.var.uri, ngx.var.args,
                                ngx.header.status)
    if not ok then
        ngx.log(ngx.ERR, "failed to create timer: ", err)
        return
    end
}

```

同样可以创建一个无限发生的时钟，示例中一个时钟通过在回调函数中再次调用 ngx.timer.at 实现每 5 秒触发一次，下面是代码：

```

local delay = 5
local handler
handler =function (premature)
    -- do some routine job in Lua just like a cron job
    if premature then
        return
    end

    local ok, err = ngx.timer.at(delay, handler)
    if not ok then
        ngx.log(ngx.ERR, "failed to create the timer: ", err)
        return
    end
end

local ok, err = ngx.timer.at(delay, handler)
if not ok then
    ngx.log(ngx.ERR, "failed to create the timer: ", err)
    return
end

```

因为时钟回调在后台运行，其运行时间不被计入客户请求应答时间，所以，它们会因为 Lua 程序中的错误或者只是单纯的负载过重就在服务端累积占用系统，耗费系统资源。为防止极端的结果，如 Nginx 服务崩溃，设置一个内建的限制，在工作进程中设置待定时钟数和运行时钟数。待定时钟数意味着还没有到期的时钟，运行时钟数指回调函数当前运行的时钟。

工作进程上的最大待定时钟数由 lua\_max\_pending\_timers 指令配置。最大工作时钟数由 lua\_max\_running\_timers 指令配置。

本函数依赖于当前实现。每一个运行中的时钟将从全局连接记录列表中占用一条记录，

配置在 `nginx.conf` 中的 `worker_connections` 指令。所以，需要确保 `worker_connections` 配置了一条足够大的值以供实际的连接或伪造连接供时钟使用（通过 `lua_max_running_timer` 配置的最大运行时钟数）。

很多 Nginx Lua API 可以在时钟回调中使用，如流 / 数据报 `cosocket`（`ngx.socket.tcp` 和 `ngx.socket.udp`）、共享内存词典（`ngx.shared.DICT`）、用户协程（`coroutine.*`）、用户轻量级线程（`ngx.thread.*`）、`ngx.exit`、`ngx.now`、`ngx.time`、`ngx.md5`、`ngx.shal_bin`，都可以使用。但是子请求 API（`ngx.location.capture`）、`ngx.req.*` API、下游输出 API（如 `ngx.say`、`ngx.print`、`ngx.flush`）是完全禁止使用的。

回调函数的参数支持多种标准的 Lua 类型（`nil`、`booleans`、`numbers`、`strings`、`tables`、`closures`、`file handles`）。有几种异常情况：不能传递通过 `coroutine.create` 和 `ngx.thread.spawn` 创建的线程对象，或 `ngx.socket.tcp`、`ngx.socket.udp`、`ngx.req.socket` 类型的 `cosocket` 对象，因为这些对象的生命周期是由请求处理器创建的，所以和处理器是相同生命周期的，而回调函数要从上下文中分开，运行在自己的上下文中。如果共享这些对象，将收到 “no co ctx found” 错误（线程对象）或 “bad request” 错误（`cosocket` 对象）。如果需要使用，在回调函数中自己创建这些对象。

### 121. ngx.timer.running\_count

语法：

```
count = ngx.timer.running_count()
```

上下文：`init_worker_by_lua*`、`set_by_lua*`、`rewrite_by_lua*`、`access_by_lua*`、`content_by_lua*`、`header_filter_by_lua*`、`body_filter_by_lua*`、`log_by_lua*`、`ngx.timer.*`、`balancer_by_lua*`、`ssl_certificate_by_lua*`、`ssl_session_fetch_by_lua*`、`ssl_session_store_by_lua*`。

说明：返回当前运行的时钟数量。

### 122. ngx.timer.pending\_count

语法：

```
count = ngx.timer.pending_count()
```

上下文：`init_worker_by_lua*`、`set_by_lua*`、`rewrite_by_lua*`、`access_by_lua*`、`content_by_lua*`、`header_filter_by_lua*`、`body_filter_by_lua*`、`log_by_lua*`、`ngx.timer.*`、`balancer_by_lua*`、`ssl_certificate_by_lua*`、`ssl_session_fetch_by_lua*`、`ssl_session_store_by_lua*`。

说明：返回待定的时钟数量。

### 123. ngx.config.subsystem

语法：

```
subsystem = ngx.config.subsystem
```

上下文：`set_by_lua*`、`rewrite_by_lua*`、`access_by_lua*`、`content_by_lua*`、`header_filter_`

by\_lua\*、body\_filter\_by\_lua\*、log\_by\_lua\*、ngx.timer.\*、init\_by\_lua\*、init\_worker\_by\_lua\*。

说明：标识当前 Lua 上下文基于哪个 Nginx 子系统。这个模块总是返回“http”。在 ngx\_stream\_lua\_module 下，返回值是“stream”。

#### 124. ngx.config.debug

语法：

```
debug = ngx.config.debug
```

上下文：set\_by\_lua\*、rewrite\_by\_lua\*、access\_by\_lua\*、content\_by\_lua\*、header\_filter\_by\_lua\*、body\_filter\_by\_lua\*、log\_by\_lua\*、ngx.timer.\*、init\_by\_lua\*、init\_worker\_by\_lua\*。

说明：返回当前 Nginx 是否是调试版本。例如，当前 Nginx 通过 ./configure option --with-debug 参数配置和编译。

#### 125. ngx.config.prefix

语法：

```
prefix = ngx.config.prefix()
```

上下文：set\_by\_lua\*、rewrite\_by\_lua\*、access\_by\_lua\*、content\_by\_lua\*、header\_filter\_by\_lua\*、body\_filter\_by\_lua\*、log\_by\_lua\*、ngx.timer.\*、init\_by\_lua\*、init\_worker\_by\_lua\*。

说明：返回 Nginx 服务前缀路径，即启动时通过 -p 参数传进入的值，或者是预编译时通过 --prefix 参数在 ./configure 脚本指定的。

#### 126. ngx.config.nginx\_version

语法：

```
ver = ngx.config.nginx_version
```

上下文：set\_by\_lua\*、rewrite\_by\_lua\*、access\_by\_lua\*、content\_by\_lua\*、header\_filter\_by\_lua\*、body\_filter\_by\_lua\*、log\_by\_lua\*、ngx.timer.\*、init\_by\_lua\*、init\_worker\_by\_lua\*。

说明：返回当前 Nginx 核心的数值型版本号。例如，版本号 1.4.3 将作为数字 1004003 返回。

#### 127. ngx.config.nginx\_configure

语法：

```
str = ngx.config.nginx_configure()
```

上下文：set\_by\_lua\*、rewrite\_by\_lua\*、access\_by\_lua\*、content\_by\_lua\*、header\_filter\_by\_lua\*、body\_filter\_by\_lua\*、log\_by\_lua\*、ngx.timer.\*、init\_by\_lua\*。

说明：返回 ./configure 命令行的参数字符串。

#### 128. ngx.config.ngx\_lua\_version

语法：

```
ver = ngx.config ngx_lua_version
```

上下文: `set_by_lua*`、`rewrite_by_lua*`、`access_by_lua*`、`content_by_lua*`、`header_filter_by_lua*`、`body_filter_by_lua*`、`log_by_lua*`、`ngx.timer.*`、`init_by_lua*`。

说明: 返回当前 `ngx_lua` 的版本号的整型值。例如, 0.9.3 版本将返回 9003。

### 129. ngx.worker.exiting

语法:

```
exiting = ngx.worker.exiting()
```

上下文: `set_by_lua*`、`rewrite_by_lua*`、`access_by_lua*`、`content_by_lua*`、`header_filter_by_lua*`、`body_filter_by_lua*`、`log_by_lua*`、`ngx.timer.*`、`init_by_lua*`、`init_worker_by_lua*`。

说明: 返回一个布尔值, 标示当前工作进程是否已经开始退出。退出发生在配置文件重载时 (HUP 信号)。

### 130. ngx.worker.pid

语法:

```
pid = ngx.worker.pid()
```

上下文: `set_by_lua*`、`rewrite_by_lua*`、`access_by_lua*`、`content_by_lua*`、`header_filter_by_lua*`、`body_filter_by_lua*`、`log_by_lua*`、`ngx.timer.*`、`init_by_lua*`、`init_worker_by_lua*`。

说明: 返回当前工作进程的 ID (PID)。这个函数比 `ngx.var.pid` 更有效, 可以用于 `ngx.var.VARIABLE` 不能使用的上下文中 (如 `init_worker_by_lua`)。

### 131. ngx.worker.count

语法:

```
count = ngx.worker.count()
```

上下文: `set_by_lua*`、`rewrite_by_lua*`、`access_by_lua*`、`content_by_lua*`、`header_filter_by_lua*`、`body_filter_by_lua*`、`log_by_lua*`、`ngx.timer.*`、`init_by_lua*`、`init_worker_by_lua*`。

说明: 返回工作进程数量 (`nginx.conf` 中 `worker_processes` 指令的值)。

### 132. ngx.worker.id

语法:

```
count = ngx.worker.id()
```

上下文: `set_by_lua*`、`rewrite_by_lua*`、`access_by_lua*`、`content_by_lua*`、`header_filter_by_lua*`、`body_filter_by_lua*`、`log_by_lua*`、`ngx.timer.*`、`init_worker_by_lua*`。

说明: 返回当前工作进程的序号 (从 0 开始)。如果总的工作进程数是  $N$ , 那么返回值将是  $0 - (N - 1)$ 。

这个函数只有在 Nginx 1.9.1+ 以上版本中返回有意义的数值, 在早期版本中返回 `nil`。

### 133. ngx.semaphore

语法:

```
local semaphore = require "ngx.semaphore"
```

说明: 实现一个典型的信号灯, 用于不同的轻线程同步。支持在不同上下文创建的轻线程中共享同一个信号灯, 条件是 `lua_code_cache` 指令打开 (默认打开) 并且所有的轻线程在同一个工作进程上。

Lua 模块不支持被模块自己装载, 但支持通过 `lua-resty-core` 库装载。具体细节可在 `lua-resty-core` 模块 `ngx.semaphore` 部分官方文档查阅。

### 134. ngx balancer

语法:

```
local balancer = require "ngx.balancer"
```

说明: 这是一个 Lua 模块, 提供一个 Lua API 允许定义一个完全动态的纯 Lua 负载均衡器。Lua 模块不支持被模块自己装载, 但支持通过 `lua-resty-core` 库装载。具体细节可在 `lua-resty-core` 模块 `ngx.balancer` 部分官方文档查阅。

### 135. ngx.ssl

语法:

```
local ssl = require "ngx.ssl"
```

说明: 本模块提供了一个方法以控制 SSL 握手, 工作在 `ssl_certificate_by_lua*` 类的上下文中。

### 136. ngx.ocsp

语法:

```
local ocsp = require "ngx.ocsp"
```

说明: 本模块提供了一个 API 执行 OCSP 请求、OCSP 应答验证和 OCSP stapling。通常, 本模块和 `ngx.ssl` 模块在上下文 `ssl_certificate_by_lua*` 中使用。

### 137. ndk.set\_var.DIRECTIVE

语法:

```
res = ndk.set_var.DIRECTIVE_NAME
```

上下文: `init_worker_by_lua*`、`set_by_lua*`、`rewrite_by_lua*`、`access_by_lua*`、`content_by_lua*`、`header_filter_by_lua*`、`body_filter_by_lua*`、`log_by_lua*`、`ngx.timer.*`、`balancer_by_lua*`、`ssl_certificate_by_lua*`、`ssl_session_fetch_by_lua*`、`ssl_session_store_by_lua*`。

说明: 这个机制允许调用其他的 C 模块通过 NDK 实现的指令。例如, 下面 `set-misc-nginx-module` 指令可以被访问: `set_quote_sql_str`、`set_quote_pgsql_str`、`set_quote_json_str`、

set\_unescape\_uri、set\_escape\_uri、set\_encode\_base32、set\_decode\_base32、set\_encode\_base64、set\_decode\_base64、set\_encode\_hex、set\_decode\_hex、set\_sha1、set\_md5。

例如：

```
local res = ndk.set_var.set_escape_uri('a/b');
-- now res == 'a%2fb'
```

同样地，encrypted-session-nginx-module 提供的指令也可以被访问：

```
set_encrypt_session
set_decrypt_session
```

详情请参阅 ngx\_devel\_kit 模块。

### 138. coroutine.create

语法：

```
co = coroutine.create(f)
```

上下文：rewrite\_by\_lua\*、access\_by\_lua\*、content\_by\_lua\*、init\_by\_lua\*、ngx.timer.\*、header\_filter\_by\_lua\*、body\_filter\_by\_lua\*、ssl\_certificate\_by\_lua\*、ssl\_session\_fetch\_by\_lua\*、ssl\_session\_store\_by\_lua\*。

说明：使用 Lua 函数创建一个用户协程，返回是协程对象；和标准的 Lua coroutine.create API 一样，但是工作在 ngx\_lua 创建协程的上下文中。

这个 API 首次使用在 0.9.2 版本开始的 init\_by\_lua\* 上下文中。

### 139. coroutine.resume

语法：

```
ok, ... = coroutine.resume(co, ...)
```

上下文：rewrite\_by\_lua\*、access\_by\_lua\*、content\_by\_lua\*、init\_by\_lua\*、ngx.timer.\*、header\_filter\_by\_lua\*、body\_filter\_by\_lua\*、ssl\_certificate\_by\_lua\*、ssl\_session\_fetch\_by\_lua\*、ssl\_session\_store\_by\_lua\*。

说明：使之前挂起的协程恢复运行；和标准的 Lua coroutine.resume API 一样，但是工作在 ngx\_lua 创建协程的上下文中。

### 140. coroutine.yield

语法：

```
... = coroutine.yield(...)
```

上下文：rewrite\_by\_lua\*、access\_by\_lua\*、content\_by\_lua\*、init\_by\_lua\*、ngx.timer.\*、header\_filter\_by\_lua\*、body\_filter\_by\_lua\*、ssl\_certificate\_by\_lua\*、ssl\_session\_fetch\_by\_lua\*、ssl\_session\_store\_by\_lua\*。

说明：挂起当前的 Lua 协程；和标准的 Lua coroutine.yield API 一样，但是工作在 ngx\_



lua 创建协程的上下文中。

#### 141. coroutine.wrap

语法:

```
co = coroutine.wrap(f)
```

上下文: `rewrite_by_lua*`、`access_by_lua*`、`content_by_lua*`、`init_by_lua*`、`ngx.timer.*`、`header_filter_by_lua*`、`body_filter_by_lua*`、`ssl_certificate_by_lua*`、`ssl_session_fetch_by_lua*`、`ssl_session_store_by_lua*`。

说明: 和标准的 Lua `coroutine.wrap` API 一样, 但是工作在 `ngx_lua` 创建协程的上下文中。

#### 142. coroutine.running

语法:

```
co = coroutine.running()
```

上下文: `rewrite_by_lua*`、`access_by_lua*`、`content_by_lua*`、`init_by_lua*`、`ngx.timer.*`、`header_filter_by_lua*`、`body_filter_by_lua*`、`ssl_certificate_by_lua*`、`ssl_session_fetch_by_lua*`、`ssl_session_store_by_lua*`。

说明: 和标准 `coroutine.running` API 一样。

#### 143. coroutine.status

语法:

```
status = coroutine.status(co)
```

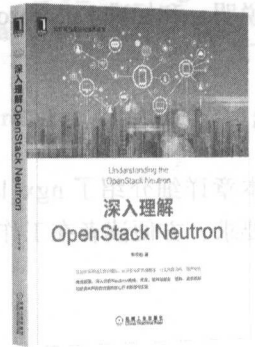
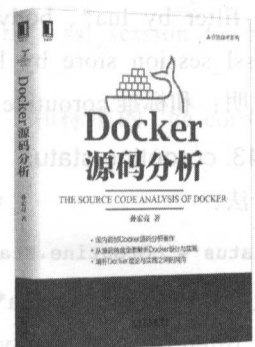
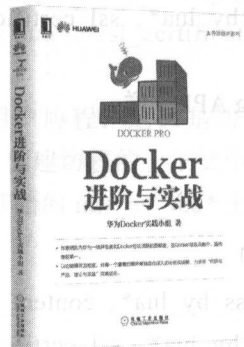
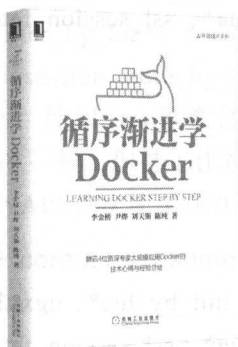
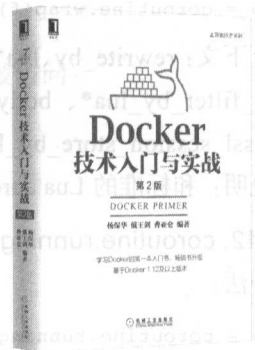
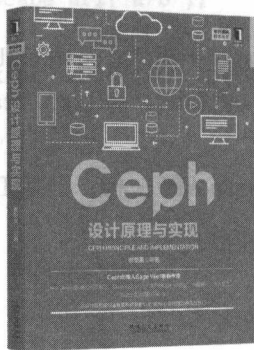
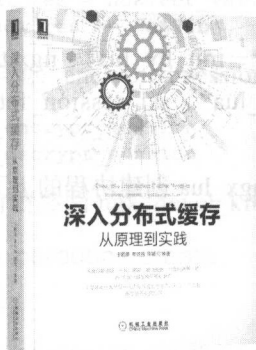
上下文: `rewrite_by_lua*`、`access_by_lua*`、`content_by_lua*`、`init_by_lua*`、`ngx.timer.*`、`header_filter_by_lua*`、`body_filter_by_lua*`、`ssl_certificate_by_lua*`、`ssl_session_fetch_by_lua*`、`ssl_session_store_by_lua*`。

说明: 和标准 Lua `coroutine.status` API 一样。

## 28.3 小结

本章详细介绍了 `ngx_lua` 模块中的 API 和常量, 并针对每一个 API 给出了语法、上下文的要求, 方便读者在工作中查阅和使用这些 API 和常量。

## 推荐阅读



## 作者简介

### 李明江

资深软件开发专家，安防领域技术专家，拥有超过20年的研发经验。创办过两家公司。

曾经在信雅达等国内多家上市公司担任研发要职，参与并主导了大量大型项目的研发。如在南方电网广州亚运会大型安保系统总体研发和管理中担任总负责人；参与中国电信全球眼规范、国家电网安保平台规范、南方电网/国家电网视频监控系统规范、公安部3111规范等规范的制定。

在C++、分布式平台开发、物联网、云计算、安防、信息安全等领域有非常深厚的积累，擅长Nginx和Lua开发相关的技术，有非常丰富的实践经验。此外，因为有多年的带领团队和创业的经历，在团队组建、团队建设、技术方向确立、核心体系搭建、核心技术攻关等方面颇有心得。

读者QQ群：196039071（Nginx Lua实战）。

Nginx上的Lua技术是近些年来由中国人章亦春整合出来的架构，将高效、轻量级的Lua脚本语言和Nginx结合起来，可以快捷、方便地开发应用系统。使用同步式的编程习惯实现异步非阻塞的高效模式，使新上手的工程师也可以快速开发高性能应用。据资深互联网专家描述以及本人亲身体会，在典型应用下，使用Lua可以使代码量减少90%左右，带来的经济效益和抢得的市场先机自不待言。对于工程师本身来讲，减少加班、多点时间喝杯咖啡也是极具吸引力的。

传统分布式平台要升级为云平台都是通过将已有的分布式平台改造成云服务，进行互联网部署，往往需要多位资深服务器开发工程师花费大量时间共同提供支持，而Nginx+Lua+Redis架构的出现，从根本上简化了这种方式，少量工程师花费少于原有体系的工作时间即可完成业务服务搭建。

Nginx+Lua架构不仅可以节约时间和成本，从做大型系统的角度来看，它还具有诸多优势：

**调试方便：**不需要编译代码，相关访问模块是成熟稳定的，只需要调试新加的业务代码即可。大型系统，特别是分布式系统，一个功能或代码的调试链条很长，非常容易出错。

**降低耦合：**因为架构的限制，Lua代码只能在必须的阶段管理器中开发，代码是一个一个.lua文件，耦合性大幅降低。

**框架良好：**先进的异步式多进程架构，可以充分利用系统资源。而如果自行开发并维护这样一个框架，则需要大量的人力、物力。

**上手容易：**Lua代码具有良好的结构和可读性，上手速度更快。团队成员经过快速培训就可以上手。

